NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

EXTENSIBLE INTEREST MANAGEMENT FOR SCALABLE PERSISTENT DISTRIBUTED VIRTUAL ENVIRONMENTS

by

Howard Allan Abrams

December 1999

Dissertation Supervisor: Michael J. Zyda

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1999

3. REPORT TYPE AND DATES COVERED
Dissertation

4. TITLE AND SUBTITLE
Extensible Interest Management for
Scalable Persistent Distributed Virtual Environments

5. FUNDING NUMBERS

6. AUTHOR(S)
Abrams, Howard Allan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(maximum 200 words)***

Eventually there will exist virtual environments inhabited by millions, but as virtual environments grow in size and number of entities, many problems emerge. Because of these problems, increasing attention is being brought to the issue of filtering data that is not of interest to a given client. Such filtering is known as interest management.

This dissertation outlines a Three-Tiered approach to interest management. The first tier breaks the world into manageable pieces. The second tier uses the data from the first to create a protocol independent perfect match between a client's interests and the environment. The third tier, building on the second, adds protocol dependence allowing the client to receive only the data from the protocol it needs. At the same time, separating out the protocol from the core interest management can allow multiple protocols to simultaneously exist within the same environment, while using the same underlying filtering mechanism.

Results from this work have shown that it is possible to create an interest management software architecture that allows bandwidth, packets per second, and CPU time to scale dependent only on the number of entities a given client is interested in at any one time.

14. SUBJECT TERMS
Simulation, Multicast, Interest Management, Distributed Virtual Environments, Bamboo

15. NUMBER OF PAGES
237

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
Unlimited

Approved for public release; distribution is unlimited

EXTENSIBLE INTEREST MANAGEMENT FOR SCALABLE PERSISTENT DISTRIBUTED VIRTUAL ENVIRONMENTS

Howard Allan Abrams
B.S., Embry-Riddle Aeronautical University, 1996

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1999

Author: ______________________
Howard Allan Abrams

Approved by: ______________________
Michael Zyda, Dissertation Supervisor
Professor of Computer Science

____
Don Brutzman
Asst. Professor of Applied Science

Rudolph P. Darken
Rudy Darken
Asst. Professor of Computer Science

____
Ted Lewis
Professor of Computer Science

Sandeep K Singhal
Sandeep Singhal, Ph.D.
IBM Corporation

Approved by: Dan C Boger
Dan Boger, Chair, Department of Computer Science

Approved by: Anthony Ciavarelli
Anthony Ciavarelli, Dean of Instruction

ABSTRACT

Eventually there will exist virtual environments inhabited by millions, but as virtual environments grow in size and number of entities, many problems emerge. Because of these problems, increasing attention is being brought to the issue of filtering data that is not of interest to a given client. Such filtering is known as *interest management.*

This dissertation outlines a Three-Tiered approach to interest management. The first tier breaks the world into manageable pieces. The second tier uses the data from the first to create a protocol independent perfect match between a client's interests and the environment. The third tier, building on the second, adds protocol dependence allowing the client to receive only the data from the protocol it needs. At the same time, separating out the protocol from the core interest management can allow multiple protocols to simultaneously exist within the same environment, while using the same underlying filtering mechanism.

Results from this work have shown that it is possible to create an interest management software architecture that allows bandwidth, packets per second, and CPU time to scale dependent only on the number of entities a given client is interested in at any one time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABE	ALSP Broadcast Emulator
ADS	Advanced Distributed Simulation
ALSP	Aggregate level simulation protocol
ANOVA	Analysis of Variance
AOI	Area of Interest
AOIM	Area of Interest Manager
BGMP	Border Gateway Multicast Protocol
CCTT	Close Combat Tactical Trainer
DIS	Distributed Interactive Simulation
DIVE	Distributed Interactive Virtual Environment
HLA	High-Level Architecture
IANA	Internet Assigned Numbers Authority
IE	Interest Expression
IGMP	Internet Group Management Protocol
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
JPSD	Joint Precision Strike Demonstration
LAN	Local Area Network

MALLOC	Multicast Address Allocation
MASSIVE	Model, Architecture and System for Spatial Interaction in Virtual Environments
MAVERIK	Manchester Virtual Environment Interface Kernel
ModSAF	Modular Semi-Automated Forces
MR	Minimal Reality
NAK	Negative Acknowledgment
PDU	Protocol Data Unit
PIM	Protocol Independent Multicast
PPS	Packets Per Second
QoS	Quality of Service
RTI	Run Time Infrastructure
Spline	Scalable Platform for Large Interactive Networked Environments
SRM	Scalable Reliable Multicast
STOW	Synthetic Theater of War
STOW-E	Synthetic Theater of War-Europe
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UFR	Update Free Region
WAVES	Waterloo Virtual Environment System

ACKNOWLEDGMENTS

Many thanks to:

- Arlene Guest, Donna Burych, and the rest of the Oceanography Department, for use of their SGI graphics laboratory to conduct my experiments.

- Mike Williams, for the use of the network equipment used to conduct my experiments.

- My dissertation committee members, Don Brutzman, Rudy Darken, Ted Lewis, Sandeep Singhal, and Mike Zyda, for allowing me the opportunity to work on this topic.

- My mother and father, for supporting my education.

- Katie, for being patient while I wrote and programmed ***every*** day for a full year.

Also, this work could not have been completed without the generous support from its sponsors: Jaron Lanier, the National Tele-Immersion Initiative, Advanced Network and Services, the Navy Modeling and Simulation Management Office, and the Office of Naval Research.

I. INTRODUCTION

A. THESIS STATEMENT

It is possible to create an interest management software architecture allowing persistent distributed virtual environments to scale dependent only on the number of entities a given client is interested in at any one time.

B. GOALS

Eventually there will exist persistent, large-scale, distributed virtual environments inhabited by millions of entities. A persistent distributed virtual environment is defined as one that is "never-ending" or "always on." This is either because its users require that it is always running, or because it is so large and distributed that stopping the entire simulation to make changes is just not possible. One persistent environment that many people are familiar with is the Internet. Pieces of the Internet are regularly upgraded, and new protocols and software are constantly being introduced. While pieces of the Internet can be rebooted, the Internet as a whole can never be taken down for upgrades. Clearly that would not be acceptable.

A large-scale distributed virtual environment is made up of many entities. In general, an entity is any independent object in the virtual environment, anything from a word-processing document to an ant to an entire city block. Each computer participating in the environment runs a specific set of software called a client, which in turn controls one or more local entities within the environment. The client is also responsible for establishing and controlling communication between itself and other remote clients. It periodically receives state information about remote entities from remote clients, and updates local information about the remote entities. Figure 1 illustrates the relationships between computers, clients, local entities, and remote entities.

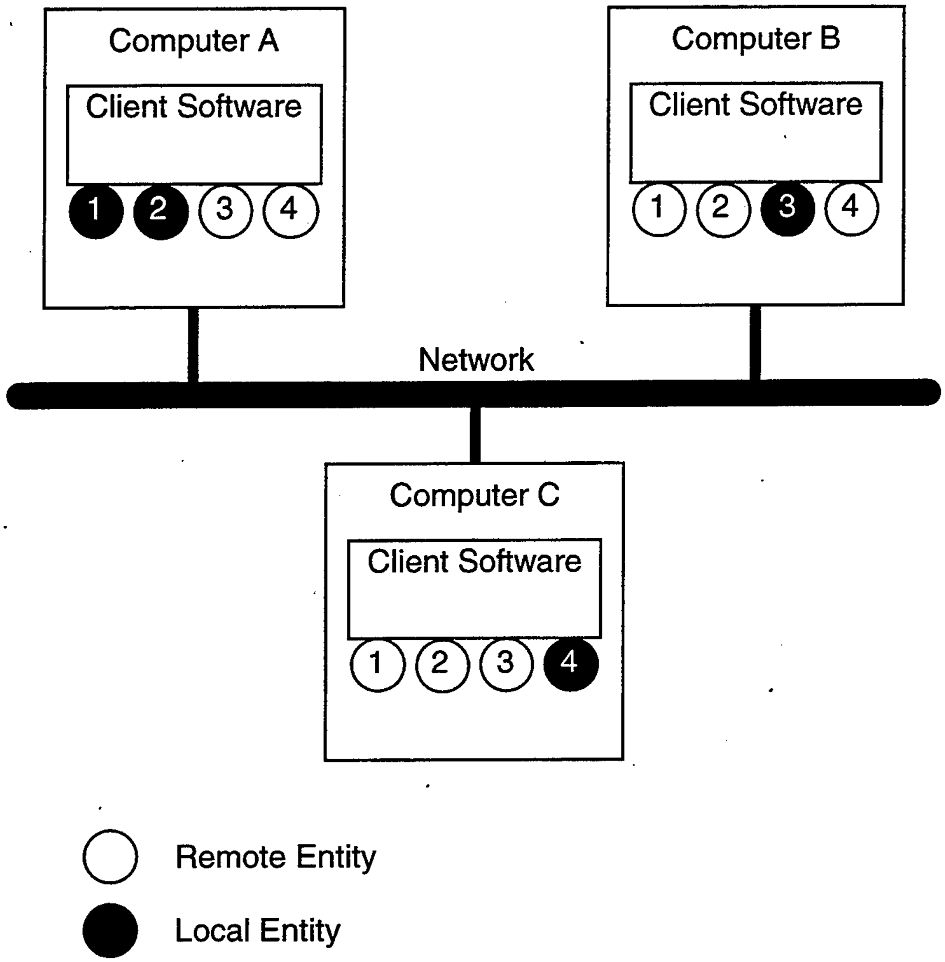


Figure 1. The relationships between computers, clients, local entities and remote entities

As virtual environments grow in size and number of entities, many problems emerge. For example, computers cannot handle receiving messages from an ever-growing number entities, and certainly cannot handle the processing of those messages in a timely manner. Past research has shown that up to 99% of all data transmitted within a pure broadcast based distributed virtual environment is irrelevant to any one local host (VanHook, Calvin et al. 1994; Morse 1999; Morse 1999). Yet in order to solve more complex problems, many researchers developing virtual environments often want or need more entities in larger, more complex environments.

To allow the virtual environment to scale, information must be filtered that is of no use to a client's local entities. This filtering is known as *interest management.* For interest management to be accomplished, entities express interest using a set of criteria known as an interest expression (IE). The interest expression criteria must be met for a remote entity to be of interest to that client. Ideally, clients only receive updates about entities that meet the IE criteria. Many researchers have tried to use interest management to reduce the effects of the problems associated with scaling, and they have succeeded to varying degrees.

The goal is to design an interest management software architecture that can allow the existence of a large distributed persistent virtual environment. That is, the goal is to create an architecture that is both scalable and extensible.

1. Scalable

To create a scalable virtual environment architecture, network bandwidth, packets per second (PPS), and CPU usage for a given client ideally independent of the total number of entities in the environment. These metrics should instead only depend on the number of entities in which that particular client is interested in at any time. Ideally this scaling should be accomplished without using prior knowledge of a particular scenario. If a virtual environment scales independently from a specific scenario, the scenario is free to change over time without compromising the environment's scalability, as long as the client is able to handle the flow of information that its local entities are interested in receiving.

2. Extensible

To make a persistent virtual environment scalable, its architecture must be extensible. That is, protocols must be able to be added and removed at runtime, without shutting down the environment. An extensible architecture called "Dynamic Protocols" (Watsen and Zyda 1998) has been developed to handle the addition or removal of application communication protocols at runtime. If one were to use a dynamically changing set of application protocols to develop a virtual environment, the interest management system used must also be extensible so that it can handle the dynamically changing set of state variables.

C. SUMMARY

A system that is both scalable and extensible would be a significant contribution to the field because a virtual environment using such a system should only be limited in scale by the number of entities in which a particular client is interested. It would also be able to be dynamically extended, and therefore will never need to be shut down for software upgrades.

D. DISSERTATION OVERVIEW

The next chapter outlines related work to this dissertation. Specifically, it describes three categories of interest management schemes and discusses their strengths and shortcomings. Descriptions of previous systems are then presented and grouped appropriately into one of the three categories. Also discussed in Chapter II is the use of Internet Protocol (IP) multicast by interest management systems, as well as its current limitations and future promises.

Chapter III presents the Three-Tier interest management system and describes how it addresses the shortcomings of the existing of interest management schemes.

Chapter IV presents the implementation decisions and details that were used in effectively realizing the design of the Three-Tier system.

Chapter V presents the experimental design and results of measurements conducted on a Local Area Network (LAN) comparing the Three-Tier system against systems using broadcast-based and region-based filtering.

Finally, Chapter VI concludes with a discussion of issues not addressed in this work, and recommendations for future work in this area.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RELATED WORK

A. OVERVIEW

This chapter presents three categories of interest management schemes and discusses their strengths and shortcomings. Descriptions of thirty-one previous systems are presented and grouped appropriately into one of the three categories. Also discussed is the use of IP multicast by interest management systems, as well as its current limitations and future promises.

B. PREVIOUS WORK

There are several existing techniques for filtering information within virtual environments. These techniques can be grouped into the following categories: server-based filtering, sender-based filtering, and region-based filtering.

The following is a description and categorization of thirty-one interest management systems. When a system appears to fit into more than one category, it is placed in the category thought to be most limiting to its scale or function. A categorized overview of these systems can be seen in Table 1 at the end of this section.

1. Server-Based Filtering

The category of server-based filtering is one in which a client that controls one or more entities connects to one or more servers. The servers decide which messages should be distributed to which clients based on each client's interest expressions. The strong point of this method is that consistency can be maintained across all the clients, and the potential for exact filtering exists because the server always has complete knowledge of the virtual environment.

There are two common problems with this method of interest management. The first is that when servers become overloaded, the only way to make the system scale is to add more servers, possibly decreasing the exactness of the filtering and/or leading to a worse form of the second problem, latency. Even in non-distributed virtual environments, latency is a major concern. If a server also acts as a packet forwarder, as is the case in most server-based virtual environments, it adds latency to message distribution. Every time a message must be forwarded, the latency increases. The larger a virtual environment that uses server-based filtering becomes, the more servers that are needed, and potentially the more times a packet must be forwarded.

In the Waterloo Virtual Environment System (WAVES) (Kazman 1993; Kazman 1993; Kazman 1995), servers called "message managers" mediate communication between hosts, and are responsible for filtering information. A host typically sends requests to the "message manager," for example, "send information which meets the following semantic criteria." These servers can also delegate point-to-point connections, for high bandwidth communications between hosts, when needed. As the size and number of entities in the virtual world increases, more servers can be added. Hosts are allocated to each server dynamically using a load balancing clustering algorithm. Because interaction detection is such a large task, specialized hosts called "Interaction detection hosts" are responsible for notifying entities of interactions. These agents continuously resolve logical constraints on objects, such as "X is near Y," using rules in a specific or general resolution mechanism. A prototype implementation called HIDRA was constructed on the Sun Sparc 2 architecture, but contained neither intelligent message filtering nor load balancing which the authors state "are necessary in large, scalable, complex virtual worlds."

RING (Funkhouser 1995) is a client-server architecture similar to that of WAVES. Entities are simulated on clients, which connect to one of many servers. Servers are connected to each other via high-speed networks, while clients can use lower speed connections to communicate to their server. Each server performs an approximate occlusion filtering of the entity updates based on pre-computed line-of-sight visibility, and only forwards these updates to other servers and clients when needed.

In BrickNet (Singh, Serra et al. 1994), clients do not communicate directly with each other but instead always communicate through servers to update each other. Communication is via User Datagram Protocol (UDP) to reduce latency. The servers in turn communicate with each other to satisfy client requests. BrickNet contains two types of objects: local and remote. Only remote objects are shared across the network. Interest management is accomplished by expressing interest in a particular object to a server. The server in turn sends the updated state information of the objects whenever it is needed.

The Distributed Interactive Virtual Environment (DIVE) system (Carlsson and Hagsand 1993; Carlsson and Hagsand 1993; Morse, Bic et al. 2000) is made up of many separated regions called "worlds." Early versions of DIVE associated each world with its own multicast address, and no interest management was performed. In later versions of DIVE each "world" relies on two servers, a "Collision Manager" and an "Aura

Manager," although communication between entities is peer-to-peer via SRM (Floyd, Jacobson et al. 1997) based reliable multicast. Entities move from world to world through gateways. An entity within a "world" queries the "Collision Manager" to determine entities and objects within visual range using the concept of "focus" and "nimbus." If something is within your "focus" you can see or hear it. If something is within your "nimbus" it can see or hear you. DIVE can be thought of as a distributed database. Different parts of the database are associated with different auras. The "Aura Managers" are used as a means to signal to the clients which parts of the database need to be replicated. Each replicated part is associated with different multicast groups. For the system to scale, a hierarchy of "Aura Managers" is used. The root "Aura Manager," known as the "Master Aura Manager," is also used as a name server.

The Model, Architecture and System for Spatial Interaction in Virtual Environments (MASSIVE) (Greenhalgh and Benford 1995; Morse, Bic et al. 2000) uses one or more servers called "Aura Managers" to determine the intersections of different entities' interests. The determination of interest is based on the collision of one or more "Auras," each representing the extent to which interaction with other entities is possible. When an intersection is found, the server tells the clients of respective entities, and the clients in turn communicate peer-to-peer until they are no longer of interest to each other. The third generation system MASSIVE-3/HIVEK (Greenhalgh 1999) which is still under initial development also uses a server for communication and filtering. To accomplish filtering it uses a system similar to Spline (see subsection 3) called "locales," where the world is broken into pieces, each hosted by a server.

Community Place, part of The Virtual Society project (Lea, Honda et al. 1997), is based on both DIVE and MASSIVE, but is targeted toward low-cost consumer equipment. Like those previous systems, each entity has a boundary called an "Aura." An entity can only be interacted with by other entities within its "Aura." A server called an "Aura Manager" is used to detect the intersection of an entity's aura with other entities within the virtual world. The server acts as a "message redistributer" which forwards messages of interest based on these intersecting auras. Multiple servers communicate via multicast to support consistency between them.

Instead of a shared database representation, AVIARY (Snowdon and West 1994) uses an object-oriented representation. Objects explicitly express interest in other objects. The common way this is accomplished within AVIARY is by using an intersection test between a viewing volume and all the objects in the world. A viewer is interested in objects that intersect its viewing volume. A server called an "Environment Database" handles this potentially large intersection test using a binary space partition tree and reports the collisions back to the objects. It is hypothesized that as the world grows multiple Environment Database servers can be used, although how to efficiently handle an intersection with multiple servers was left as future work.

In the Joint Precision Strike Demonstration (JPSD) (Powell, Mellon et al. 1996; Morse, Bic et al. 2000) each simulation is connected to a sever called a "Run Time Gateway" via an ATM IP Multicast virtual LAN. "Run Time Gateways" in turn communicate with each other using several multicast addresses. One multicast address is used for interest expressions, and several are used for data. Packets are sent in the form of

Protocol Data Units (PDUs). Entities send interest expressions to the "Run Time Gateways" in terms of PDU type and a limited set of derived fields that are recalculated for each incoming PDU. The "Run Time Gateways" act as a filter of outgoing and incoming PDUs, and only passes PDUs that conform to one of its currently active interest expressions. If a PDU is of interest to other, non-local, entities, the PDU is passed to other "Run Time Gateways" with matching interest expressions. Interest expression evaluation is performed using a predicate evaluation framework that allows dynamic, application-specific predicates to be executed using application-specific compiled-code.

NetEffect (Das, Singh et al. 1997) uses a client-server architecture. In each world, there is a master server that is connected to many peer servers through point-to-point connections. The virtual world is broken into many "communities." Each peer server is responsible for one or more communities that have been delegated to it by the master server. Clients connect to one peer server at a time, but can migrate to another server at any time. All clients participating in a particular community are connected to the same server to eliminate inter-server communication. Peer servers send object updates to clients in a "need-to-know" fashion. This is done for objects "in the place which the user is located," for example only objects located in the same room as the client. All client communication is via a direct Transmission Control Protocol (TCP)/IP connection with a peer server with the exception of voice chat, which is handled directly between clients. To accomplish load balancing, communities and clients can be migrated from a peer server with a high load to one with a lower load.

Systems using the Aggregate Level Simulation Protocol (ALSP) (Wilson and Weatherly 1994; Morse, Bic et al. 2000) use a hierarchy of servers to tie together many architecturally dissimilar simulations. Each server, called a "ALSP Broadcast Emulator" (ABE), is connected to other ABEs or to a simulation. Original versions of the ABEs were nothing more than packet forwarders. Messages would be received on one network connection and sent out on all others. ALSP eventually had problems scaling, and interest management was added to the ABEs. In the newer versions, objects and events are filtered at the sender, while messages are dropped at the receiver based on attribute values within the messages. All current ALSP systems are in the process of being converted over to using High-Level Architecture (HLA) (see subsection 3).

The Manchester Virtual Environment Interface Kernel (MAVERIK)/Deva (Hubbold, Dongbo et al. 1996; Pettifer 1997; Pettifer 1999; Morse, Bic et al. 2000) shares the same vision as this dissertation in that it tries to support a persistent virtual environment that does not ever need to be rebooted. However, each environment uses a server that performs all of the interest management within that environment. In an attempt to make the system scalable, each server task can be distributed among many processors. Each client connects to the server using TCP/IP, and communicates through a software layer called a "Spatial Manager." Each environment can select its own "Spatial Manager" to best fit its needs. Spatial Managers can support various types of filtering including grid cells and n-dimensional trees, but currently each client receives a unicast copy of every message the server-based Spatial Manager receives. However future designers can write custom "Spatial Managers" and install them in a server. This means that future versions of MAVERIK/Deva can support various types of server based interest management.

V-Worlds(Vellon, Marple et al. 1998) uses a single server that accepts connections from many clients. The world is partitioned into "rooms," and a client can only be in one room at a time. To limited communication the server uses a "bystander" algorithm which only updates objects contained within the clients current room and are not contained within closed containers. Clients can also register explicit interest in an object, although the authors discourage this practice.

Out of all of the above systems, only MAVERIK/Deva is extensible at runtime, and all are classic client-server systems, and will therefore have problems scaling due to latency problems with the exception of DIVE, MASSSIVE, and JPSD. Each "world" in DIVE and MASSIVE uses a server which must perform a O(N^2) collision problem between each entity within the "world", clearly this is not scalable. Not only does JPSD use gateways that add latency, it also has problems scaling because it relies on sender-based filtering, which is described in the following subsection.

2. Sender-Based Filtering

The category of sender-based filtering is one in which the sending entity decides which other entities need to receive a message. The strong point of this category of filtering is that remote clients only receive messages of interest, and never need to throw away information. One common problem with this scheme is that a sending entity needs to have knowledge of, as well as evaluate, the interest expressions of all other entities in the simulation. In these systems, scale is clearly dependent on the total number of entities within the environment. Also, if these interest expressions include relative location, then the sending entity must also know the positions of all other entities in the simulation. In the worst case if all sending entities are receiving entities, then every entity may need to regularly receive information about every other entity, altogether defeating the purpose of interest management. Another problem with sender-based filtering is that the sending entity has two choices on message delivery, the first being sending a copy of each message to every entity interested, increasing bandwidth usage and latency. The other choice is to dynamically group receiving entities with similar interests, which also increases latency and bandwidth, can be costly in terms of CPU time, and in general yields a less optimal result (Levine, Crowcroft et al. 1999).

The Minimal Reality (MR) Toolkit (Shaw, Green et al. 1993; Wang, Green et al. 1995) was extended through its Peer package to support multiple user networking. Each recipient maintains a list of other participants or “peers” to which it is connected. By default each peer is connected to every other peer using UDP messaging. The toolkit only sends messages for shared variables that have changed. Messages may be sent to any or all peers at once. Sender-based interest management is accomplished by sending messages to select peers. Receiving peers can also request other peers to stop or start sending messages to them for a period of time by sending “quenching” or “unquenching” messages.

All entities in Advanced Distributed Simulation (ADS) (Mellon and West 1995) access data about other entities through a “Ground Truth Database,” which is shared across all hosts in the simulation. Each host maintains its own copy of the database, which is minimally updated based on the needs of the entities it is simulating. After entities declare their interests to the "Ground Truth Database," the database itself is responsible for maintaining the set of data items in the database that match the union of all the entities interests. The interest expressions are predicate-based, and are evaluated at all the sources. The interest expressions also support variable resolution data, which also must be calculated at the source. To reduce overall computation at the source, the authors suggest specifying a small set of resolution parameters, and combining predicates. The authors also suggest that applying a scheme such as grid based filtering is possible by adding fields to items in the database. These fields can then be used as a first order pass filter.

A recent approach (Morse 1998; Morse 1999; Morse 1999) still under investigation uses mobile agents to create dynamic multicast groupings using a new distributed algorithm. It is to be built on top of HLA (see subsection 3) and its goal is to dynamically group senders and receivers based on their interests and transmissions. Once senders and receivers are grouped, they can communicate using one of a fixed number of multicast groups. It is hoped that this dynamic grouping offers a good trade off between fixed region based filtering, and a multicast per entity approach.

None of the above systems are extensible at runtime, and all are classic sender-based systems that require total knowledge of all entity interest expressions with the exception of the recent work on dynamic multicast grouping. Because this approach is still under investigation, how well it scales is still unknown, but it will certainly yield increasingly less exact filtering as the number of entities increases.

3. Region-Based Filtering

The category of region-based filtering is one in which the virtual environment is statically broken up into pieces or regions. The tesselation of the world allows for rough filtering based on an entity's location. However, these regions need not be limited to the three-dimensional space of location, in some systems they can be of arbitrary dimensions and shape. Typically, sending entities send messages to a multicast address assigned to a region, while receiving entities subscribe to the multicast addresses of regions of interest. One strong point of this scheme is that deciding which regions to send to, and which regions to subscribe to, does not consume many CPU cycles. Another strong point is that by using multicast most of the filtering is done by the network hardware, not by the client's software. One common problem is that because this is only a rough filtering, more information may be received and processed than is needed.

Another problem is that a virtual environment can suffer from what is known as crowding or clumping. If many entities crowd, or clump, into one region, an entity which subscribes to that region may be overwhelmed by entities it cares little about. In its worst form, the clumping problem degrades the system to a simple broadcast communication paradigm.

To avoid this clumping problem, past research has partially focused on determining the "right" size for regions. One researcher determined that a region size of 2 to 2.5 km provided the most significant reduction in total host bandwidth usage (Rak and VanHook 1996). Yet another researcher concluded that hexagons of size 4 km, approximately equal in area to 6.5 km square regions, were optimal (Macedonia, Zyda et al. 1995). Clearly both answers are different, yet both have supporting experimental data proving that they are correct. This indicates that optimal region size is a function of the application scenario, and that one answer cannot be obtained.

Other research to avoid this clumping problem focused on dynamically subdividing regions in an attempt to create an even distribution of entities per region. One particular researcher was even able to show that an $O(N^2)$ collision detection simulation can be solved in $O(N)$ time regardless of entity distribution using dynamic partitioning of the non-distributed environment (Harless and Rogers 1995).

In Space Fusion (Sugano, Otani et al. 1997) the world is broken into many regions. Each region can contain information about a different geographical area, or different information for the same geographical area. The mode of communication is a client-server architecture in which a client connects to several servers at once and then "fuses" information from many regions together. The client also calculates which entities it is interested in based on the locations of entities within the regions, and the server that contains the entity sends detailed data to the client when needed.

A prototype system for GreenSpace (Pulkka 1995) used a combination of grid based filtering and the COMIC spatial model of interaction that is also used by DIVE and MASSIVE. Each cell in a static grid is assigned a multicast address, and an entity in the world subscribes to addresses for the grid it is in as well as the eight cells directly adjacent to it. Entities send messages to the address of the cell they are currently in and send messages to a "group update" address when they change cells or have been in the same cell for a long period of time. Two types of messages are supported: text and position. Position messages are used to render entities within the virtual world. Entities are rendered in an estimated position when detailed information is not available because those entities are not within one of the nine cells currently subscribed to. When an entity receives a text message, it is ignored if it is from another entity more than one cell away. If the sending entity is less than a cell away but more that a half cell away, the system simulates an audio drop-off by replacing the text of the message with "UserX is saying something, but is too far away to hear."

The most sophisticated interest management that Modular Semi-Automated Forces (ModSAF) (Smith, Russo et al. 1995; Morse, Bic et al. 2000) has used was in version 1.4. That particular version included extensions for cell based multicast support. Using these extensions, the gaming area is broken into a grid of cells. Each cell has two multicast addresses, one for low fidelity and one for high fidelity. Packets in the form of PDUs are sent to the multicast address of an individual cell based on the location of the event, or entity. The high fidelity address is used if an entity has already sent an entity state PDU to the low fidelity address within some time threshold. The low fidelity address is used otherwise, as well as for all other types of PDUs. Entities express interest by subscribing to either the low fidelity address or to both the low and high fidelity addresses. The use of two multicast address per cell reduces the amount of data that wide area viewers, such as plan view displays, have to receive. Although this helped to reduce traffic, each wide area viewer has its own idea of what "low fidelity" means; the authors state that "...this concept needs to be further refined." (Smith, Russo et al. 1995)

The Close Combat Tactical Trainer (CCTT) (Mastagllio and Callahan 1995; Morse, Bic et al. 2000) was built on top of a FDDI network using a combination of grid-based multicast groups and filtering based on PDU types. Entities subscribe to multiple grid addresses based on their areas of interest and then throw away incoming PDUs that do not fit their interest expression.

Proximity Detection (Steinman and Wieland 1994; Morse, Bic et al. 2000) uses a "fuzzy"-grid-based filtering mechanism. The grid cells are considered "fuzzy" because the radius of the area of interest of each entity is extended to take into account some uncertainty in entity positions. Each entity receives messages for all other entities within its fuzzy grids of interest. The receiving entity may then drop messages based on its specific interests.

In (Macedonia, Zyda et al. 1994; Macedonia 1995; Macedonia, Zyda et al. 1995), the Area of Interest Manager (AOIM) for NPSNET-IV is described. Although this AOIM was never implemented, a simulation was developed to predict and evaluate the results if such an AOIM was used within NPSNET-IV. The AOIM divides the gaming area into a grid of hexagonal cells. Each cell is mapped to a multicast group. An entity expresses its interest by subscribing to the multicast addresses of cells that intersect its area of interest. When an entity first subscribes to a cell, it issues a join PDU to that cell. The oldest entity in that cell responds via TCP/IP with a PDU containing the aggregation of the entity state PDUs for all entities within that cell. When an entity leaves a cell, it issues a leave PDU. In standard Distributed Interactive Simulation (DIS), entities are required to transmit their state PDUs periodically, typically every 5 seconds, even if no state information has changed. NPSNET-IV's addition of simple join/leave functionality to the DIS protocol allows the designers of virtual environments to remove this retransmission requirement, which greatly reduces bandwidth consumption.

The Synthetic Theater of War-Europe (STOW-E) (VanHook, Calvin et al. 1994; VanHook, Rak et al. 1994; Morse, Bic et al. 2000) also divides the gaming area into grids. Each local area network has an Application Gateway connecting it to the wide area network (WAN). A LAN's Application Gateway is responsible for communicating the interests of entities on its LAN. It does this by sending the union of grid cells that each entity is interested in to all the remote Application Gateways. When a PDU from an entity on a local LAN falls within one of the cells that it received from a remote gateway, the Application Gateway broadcasts the PDU to all the other application gateways. PDUs that fall outside one of the requested cells are broadcast only once in every n PDUs, as a lower fidelity transmission.

Using a scheme similar to ModSAF 1.4, STOW ED-1 (Calvin, Cebula et al. 1995; Morse, Bic et al. 2000) uses a two grid, with each grid cell having two multicast addresses. An entity expresses interest of a cell at low fidelity by subscribing to one of the two multicast addresses and at high fidelity by subscribing to both addresses. Instead of subscribing to the multicast addresses themselves, the entities send their interest expressions to a host on the LAN called an Agent Host. The Agent Host is then responsible for subscribing to the requested multicast addresses. The data from these groups is then broadcast back to the LAN.

One unique system (Makbily, Gotsman et al. 1999) uses concept of "update free regions" (UFRs). A UFR is a region that is calculated by each entity based on geometric occlusion. While the entity is contained within its UFR, it does not need to send state updates to other entities. Whenever an entity leaves its UFR, it transmits a state message, and recalculates the extents of its UFR. This provides a very good filtering, but has two drawbacks. First, for a given entity a UFR must be calculated for every other entity in the environment. Second, the algorithm to calculate a UFR is $O(N^3)$ with respect to the number of occluder vertices in the environment.

Another unique approach (Lim and Lee 1999) divides regions into sub-regions which border adjacent regions. Instead of being aware of multiple regions, an entity is only aware of the region it is in and the sub-regions of the adjacent regions that border its region. This is thought to help a distributed virtual environment scale because entities need not have full knowledge of neighboring regions. The size of a given sub-region can be dynamically controlled based on entity density within a region. Regions with higher entity counts would have smaller sub-regions, while regions with low entity counts would have large sub-regions.

In Scalable Platform for Large Interactive Networked Environments (Spline) (Barrus, Waters et al. 1996; Waters, Anderson et al. 1997), the world is broken up into regions in space called "locales." A region-based communication server handles the communication for one or more "locales," and each locale can dynamically change servers at any time. Objects in the virtual world must reside in only one "locale." A process expresses interest by placing special objects called "spCom objects" in each "locale" it is interested in. A process will then receive updates via multicast for every object in those "locales" of interest. Also controlling the virtual world are content-based communication servers. Each serves many "Beacons," which are objects that contain URL-like tags. These "Beacons" are used to locate "locales" of interest based on content, and the content-based servers act as a sort of dynamically changing name server for locales.

SmallView (Broll 1997) breaks up the virtual world into regions. Each region has arbitrary shape and has its own multicast group. Regions are made up of three parts: radiation, horizon, and hull. Clients that are within the radiation portion of the region subscribe to its multicast group. By subscribing to the multicast group, the client will receive messages about things that happen within the horizon portion of the region. Clients outside the radiation, but within the hull see only an external representation of the region. Clients that are not multicast capable can use a unicast address for communication instead. To make multicast more reliable, a server is used for acknowledgment. Clients that receive an acknowledgment, but did not receive the actual data can request it from the same server via unicast.

MASSIVE-2 (Greenhalgh and Benford 1997; Morse, Bic et al. 2000), like its predecessor MASSIVE, uses the concept of auras to represent the extent to which interaction with other entities is possible. However, MASSIVE-2 uses arbitrarily shaped, mobile regions called "third-party objects" to help mediate communications. These regions can change the relationship between objects by introducing notions such as attenuation. Regions can also be used to represent an aggregate version of its contents when clients are farther away. Each region is assigned a multicast address. When an entity's spatial interest intersects with a region's extents, the entity subscribes to that region's multicast group. When no intersections occur, the entity subscribes to a default multicast address for the world.

The U.S. Defense Department High-Level Architecture (HLA) (Dahmann, Weatherly et al. 1999; VanHook, Rak et al. 1996; Morse 1998; Morse, Bic et al. 2000) uses a generic filtering scheme called "Routing Spaces." A routing space is a multi-dimensional attribute space. For example a location vector maps to a three-dimensional "routing space," or a temperature scalar maps to a one-dimensional "routing space." An entity expresses interest in sections of a "routing space," for example a volume within a three-dimensional location "routing space." These sections of interest are called "Subscription Regions." Entities also update within regions of the "routing space." These regions are called "Update Regions." Connections are made between entities with overlapping subscription and update regions. How these connections are made is implementation dependent. The current standard implementation, called the Run Time Infrastructure (RTI) version 1.3, statically breaks up a "routing space" into sections, and assigns a multicast address to each section. Entities simply subscribe to the multicast addresses correlating to the sections of the routing space they are interested in. As entities change interests, they change multicast groups. For a location vector, this process maps exactly into a three-dimensional grid based filtering scheme.

Projection Aggregation (Singhal and Cherition 1996) uses a combination of hierarchical grid based filtering along with aggregation of entities. Like many of the schemes above the world is broken up into regions. However, each region can have multiple addresses, called projections, one for each different type of entity. This is similar to a multidimensional attribute space in HLA. For example, tanks transmit to one address, while cars transmit to another, even though they are within the same region. Also within the virtual environment are logical entities, called "Organization Aggregations," which represent an aggregate version of many entities, within many regions. An entity then subscribes to aggregate projections of regions depending on its interests.

None of the above systems are extensible at runtime, and with the exception of UFRs, MASSIVE-2 and Projection Aggregation, all use static regions that suffer from the clumping problem. UFRs have clear scalability problems because of the time complexity of the algorithm. MASSIVE-2 added support for mobile regions, however these regions must transmit their positions to a single default multicast address, leading to a different form of the clumping problem. Although Projection Aggregation uses an octree, it may still be possible in some cases for it to suffer from a form of the clumping problem.

4. Summary

Table 1 presents a summary of filtering schemes used by the thirty-one past systems described above. Although it does not fit easily within this categorization, the Three-Tier system described in the next chapter is listed at the bottom.

	Server-based	Sender-based	Region-based
WAVES	✔		
DIVE	✔		
AVIARY	✔		
BrickNet	✔		
MASSIVE	✔		
MASSIVE-3	✔		+
Community Place	✔		
RING	✔		
JPSD	✔	+	
NetEffect	✔		
ALSP	✔		
MAVERIK/Deva	✔	∗	∗
V-Worlds	✔		+
MR Toolkit		✔	
ADS		✔	∗
Dynamic Multicast Groups		✔	
Space Fusion	+		✔
GreenSpace			✔
ModSAF 1.4			✔
CCTT			✔
Proximity Detection			✔
NPSNET			✔
STOW-E	+		✔
STOW ED-1	+		✔
UFRs			✔
Sub-regions			✔
Spline			✔
SmallView	+		✔
MASSIVE-2			✔
HLA (RTI 1.3)	◇	◇	✔
Projection Aggregation			✔
Three-Tier system			✔*

✔	System uses category as its primary interest management
+	System uses category as its secondary interest management
◇	System has the potential to use category instead of current interest management
∗	System has the potential to add category as secondary interest management
✔*	System uses a hybrid hierarchical region/protocol independent/protocol dependent filtering

Table 1. Summary of filtering schemes used by past systems

C. MULTICAST AND ITS LIMITATIONS

Many past and current interest management systems use IP multicast to accomplish implicit filtering by the hardware within the network infrastructure. The system presented in this dissertation relies heavily on future IP multicast scalablity, so it is important that the reader is aware of its problems and promises.

IP multicast is many-to-many network delivery mechanism based on the UDP datagram. Senders of IP multicast packets do not send to a particular Internet address, but instead send the packet to a group address. The 268 million IP version 4 (IPv4) Class D addresses 224.0.0.0 through 239.255.255.255 are reserved as multicast group addresses (Stevens 1998). An advantage to multicast is that because the packets are addressed to a group rather than to a specific receiver, the sender never needs to know the set of receivers, so it may be arbitrary in number. Another advantage is that only one copy of the packet needs to be transmitted by the sender no matter how many receivers are listening. Receivers open a network connection similarly to receiving broadcast traffic but also must "join" the multicast groups that they are interested in. Joining is achieved by sending an Internet Group Management Protocol (IGMP) packet to a well-known multicast address. This 'join' packet informs routers listening on this address that the sender would like packets from a certain multicast group forwarded on to the local network segment. The result is something similar to radio. Senders transmit on many different channels, and receivers tune their radios to the channels they are interested in. Although there is nothing inherently limiting in the IP multicast architecture, there are many problems and limitations with the current implementation on the Internet.

1. Address Space Limitations

Like many emerging technologies, the current implementation of multicast on the Internet has limited capacity. One limitation of multicast is the amount of address space available, and how it is allocated. Under the IPv4 mapping to Ethernet, the address space for multicast is limited to just over 8 million addresses (Stevens 1998). Although this is a large number, if each virtual environment were to use thousands or millions of address, this resource may quickly run dry.

Already IP version 6 (IPv6) is being used on the Internet in the form of the 6Bone (Durand and Buclin 1999; Fink 1999), an IPv6 network tunneled over the regular Internet. Under the current IPv6 mapping to Ethernet, the address space used for multicast routing is currently over 32 bits, but has allocated space for over 112 bits to be used once routing hardware and software catch up (Stevens 1998). IPv6 is designed to run side by side with its predecessor IPv4, allowing a slow but easy migration to a native IPv6 Internet. Recently the Internet Assigned Numbers Authority (IANA) has even begun to allocate IPv6 address, however the complete IPv6 transition of the entire Internet is not expected until at least the year 2005 (Kahney 1999).

2. Address Allocation Problems

Although 112 bits of space will likely provide enough addresses for any need, there is currently no way of dynamically allocating an address for a specific purpose. One proposed solution to this problem is the Multicast Address Allocation (MALLOC) architecture (Handley, Thaler et al. 1999). The MALLOC architecture is a three-layer set of protocols, along with their implementations in the form of server and client software.

Put simply, MALLOC allows a software application to reserve a set of multicast addresses for a specified length of time. This architecture is currently under testing, and as of October 1999, an alpha implementation had already become available to the general public.

3. Router Configuration Problems

Even if a software application has a set of addresses reserved for its use, there is no guarantee that IP multicast will work properly because most routers on the Internet are not currently configured for IP multicast. The *Multicast Debugging Handbook* (Thaler and Aboba 1998) outlines several common symptoms of mis-configured routers. A partial list includes packet loss, duplicate packets, and receiving packets from only some multicast groups. Currently the only general method to get multicast traffic across the vast Internet is via the MBone (Macedonia and Brutzman 1994; Casner, Kristol et al. 1997), a tunneled network similar in concept to the 6Bone.

Even if two applications have every router between them configured for multicast, some of their dynamically reserved multicast addresses might not work. When a router is configured for multicast, it has a limit on how many multicast address routes it can hold in memory. Because in many cases this constraint is simply a memory issue, it is very easy to add support for more addresses. For example if a router requires 1K of memory for each multicast route, one can simply configure the router with 8 gigabytes of memory to support the entire IPv4 Ethernet multicast address space. This is a potentially viable solution for IPv4, but not for IPv6. If a router requires 1K of memory for each multicast address, it would need over $2x10^{115}$ bytes of memory to support the entire IPv6 address space. Clearly this solution is not viable for the foreseeable future. One possible solution to this problem is Border Gateway Multicast Protocol (BGMP) (Thaler, Estrin et al. 1998), which works closely with the MALLOC architecture to provide multicast routing. BGMP creates shared distribution trees with the root located anywhere within an entire single domain. Which domain is used is decided based on which MALLOC domain the multicast address was allocated from. Because BGMP uses shared trees, it is optimal for single source multicast groups which are created by the group-per-entity paradigm made employed the architecture to be presented in Chapter III. The authors of BGMP have outlined a transition strategy for the Internet, but it is not clear how long this process would take. Other routing protocols that also may solve this problem are hierarchical Protocol Independent Multicast (PIM), and bi-direction PIM with GRIB (Oran 1999). In the end, which protocols become standard is ultimately up to the large router vendors to decide.

4. Network Interface Card Problems

Even if all of the routers between two applications are able to handle all of the reserved multicast addresses, there is still one hardware hurdle left. The network interface cards on most computers are extremely limited in their multicast capability. The most extensive support on PC cards today is a 512-bit hash table, allowing partial hardware filtering of up to 1024 multicast groups (McGregor 1998), and many cards support fewer addresses. If any host must subscribe to more multicast groups then its network card supports then the filtering of multicast packets must take place completely in software within the network layer of your operating system. Recent research shows that when there is a lack of hardware support, up to 10% of a modern CPU can be required to filter a 10Mbps network saturated with multicast packets (McGregor 1998). While this result is disappointing, it may not be limiting to many applications. In the worst case, the most demanding applications need only add a second current day $200 CPU to their machines making them capable of filtering a multicast saturated 100Mbps network. If CPU speeds increase at the same rate as network bandwidth a two CPU computer will always be able to filter a completely saturated network LAN, leaving one CPU free.

The above numbers represent a worst case. Recall that the multicast filtering that the network interface card must accomplish is, in essence, a second stage filter because the multicast router only forwards packets onto the LAN that hosts on that LAN requested. If there is only one host on the LAN using multicast, then the network layer will only have to receive and filter relevant traffic, which it would have to do anyway.

5. Unreliable Nature

Because senders cannot know to whom they are sending, they can never know whether the message they sent was received. This is why IP multicast is unreliable by nature. Several attempts have been made to devise schemes that can overcome the unreliable nature of multicast. The Scalable Reliable Multicast (SRM) architecture uses sequence numbers, and negative acknowledgments (NAK) with statistical random timers to provide a scalable framework for large-scale multicast applications (Floyd, Jacobson et al. 1997). Other approaches include various combinations of request/repair, hierarchical approaches, acknowledgments, negative acknowledgments, aggregated negative acknowledgments, statistically based negative acknowledgments, polling, forward error correction, all of which use multicast and/or unicast to handle out of band communication (Multicast-Implementation-STudy 1999). While these "reliable" multicast protocols work in many situations, it is difficult to guarantee that a single non-sequenced packet will reach all of its destinations.

One future development on the Internet that will help with this particular problem with multicast is the concept of Quality of Service (QoS). QoS allows a router to guarantee network qualities such as latency, jitter, and bandwidth. Although there are several competing systems for handling QoS (Stardust.com 1999), the leading research seems to be focused on "differentiated services." Differentiated services uses 4 bits within a packet's header to differentiate that packet from other packets. Depending on which bits are set, a packet receives a different quality of service. The highest quality of service is sometimes called "Premium" or "Virtual Leased Line." This "Premium" service guarantees, among other attributes, an amount of bandwidth. As long as a stream stays within its bandwidth limits, its packets will never get dropped due to router congestion. Although how to apply QoS to multicast is still an open issue, this bandwidth part of the "Premium" service would prevent multicast packets from being dropped due to congestion, making multicast much more reliable without the need for supporting reliability protocols. QoS is currently being tested on the Internet. Developers and testers have formed a test on the Internet call the QBone (Teitelbaum 1999). The QBone, although similar in name to the MBone or 6Bone, is not a tunneled network over the regular Internet. Instead QBone is simply a contiguous set of networks within the Internet each providing interdomain "Premium" QoS.

Another unreliable attribute of multicast is that when a host requests to join a multicast group, an uncertain amount of time will elapse before the host receives data from the multicast group. This has become known as "time to join" for a multicast group. When a computer joins a multicast group, it sends an IGMP message to the nearest router. This router then subscribes to the multicast group by sending messages to other neighboring routers. This process continues along this new branch of the multicast group's distribution tree until it is grafted onto the main tree. Time to join the group is therefore the time to propagate the subscription request up to the nearest existing branch of the distribution tree, so that multicast packets can be routed to the newly-subscribed host. In the worst case, this delay is on the same order of magnitude as the round-trip time to the root of the multicast distribution tree. This is not an attribute that is going to improve much, but one can use this knowledge to plan ahead when subscribing to multicast groups. By an anticipating multicast subscriptions, an application may be able to lessen the impact of these delays.

6. Summary

Although the current multicast implementation on the Internet has its limitations, research is well on its way toward supporting applications that require a large number of multicast addresses. Several aspects of this dissertation assume success from these current research efforts.

D. SUMMARY

In this chapter three categories of interest management schemes were described and their strengths and shortcomings discussed. Descriptions of thirty-one previous systems were presented and grouped appropriately into one of the three categories. Also discussed was the use of IP multicast by interest management systems, along with its current limitations and future promises. The next chapter presents the theory and design of the Three-Tier interest management system, a solution that attempts to avoid the shortcomings of these three categories of interest management schemes while still retaining their strengths.

III.DESIGN OF THE THREE-TIERED INTEREST MANAGEMENT SYSTEM

A. INTRODUCTION

This chapter presents the theory and design of the Three-Tier interest management system. Specifically it outlines the overall goals for a successful interest management system, and the theory behind each of the three tiers in the system presented in this dissertation.

B. GOALS

For an interest management system to help a virtual environment become completely scalable it must overcome the shortcomings of the three categories of interest management. That is, the system must not have high latency and therefore behavior communication must be server-less during active participation. It must not suffer from the "clumping" problem, where entities can easily crowd into one region. It should use multicast to take advantage of filtering at the network hardware level, while at the same time avoid redundant communication. It should use regions to break the virtual environment into manageable pieces, yet allow for near perfect filtering according to clients' information needs. These regions should be hierarchical, to allow for efficient aggregation. It should be extensible, allowing multiple application protocols to be dynamically added or removed from the system at run-time.

C. OVERVIEW

Interest management has typically been a one-step process. Data flows in from the network, and then interest manager software roughly filters it, hopefully passing what would be mostly relevant data on to the client. This dissertation outlines a three-step, or tiered, approach to interest management. The first tier uses region-based filtering, where the world is divided into manageable pieces. However, these approaches are extended to allow the regions to change size dynamically, thereby eliminating the clumping problem. Also, the information sent to these regions is only sent at a low rate and contains low fidelity information, thereby reducing the amount of information that must be transmitted and processed. The goal of the first tier is to receive and process just enough information to allow the second tier to determine whether an entity is of possible interest at any particular moment in time.

The second tier uses the data from the first tier to create an application protocol-independent perfect match between a client's interests and the environment. This second pass is done in a broad and protocol-independent manner using interest expressions common to most virtual environment applications.

The third tier, building on the first two, adds application protocol dependence, potentially allowing the client to receive only the needed data from the protocol it. At the same time, separating the protocol from the core interest management can allow multiple protocols to simultaneously exist within the same environment, while using the same underlying filtering mechanism. Figure 2 shows a simple dataflow diagram of these tiers.

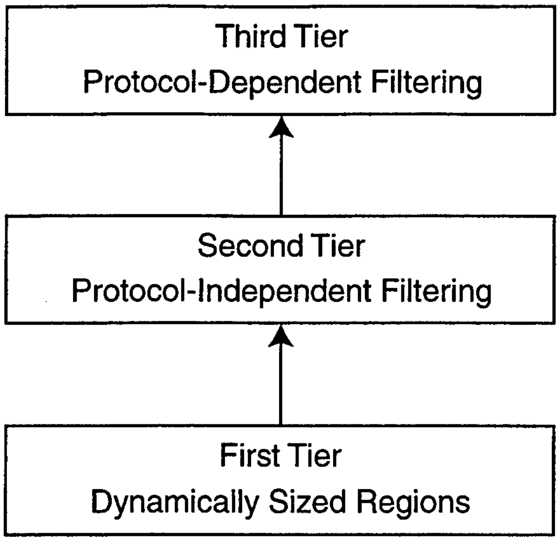

Third Tier
Protocol-Dependent Filtering
Second Tier
Protocol-Independent Filtering
First Tier
Dynamically Sized Regions

Figure 2. Dataflow Between Tiers

It is proposed that, together, these tiers create an excellent match between what the client wants to receive, and what it actually receives, while conserving network bandwidth and CPU cycles.

D. THREE-TIER DESIGN

1. Tier One: Dynamically Sized Regions

Data is often only roughly filtered by interest management systems that handle large numbers of clients. The reason for this is that it can be too costly in terms of both CPU time and bandwidth to calculate an exact intersection of a client's interest expression (IE) and all of the data arriving over the network.

However, if the roughly filtered data was used as a first pass toward computing this intersection, it may be possible to compute exactly the data needed for a given client by choosing from only this subset of the available data. It is hypothesized that by reducing the dataset in this manner, the size of a simulation would only be limited by the number of entities a given client is interacting with, not the number of entities in the entire simulation.

One measure present in most IEs is the distance from an entity. Typically this area of interest (AOI) can be represented by a sphere with a radius equal to the maximum distance of interest. As described in Chapter II, several approaches have used spatially based multicast groups to reduce network and CPU load at a hardware level. The success of these approaches is defined by the size of each region. If a region is too small, a client has to subscribe to too many groups, and entities switch multicast regions frequently. If the region is too large, a client has to listen to other entities that it did not care about, and in the worse case, the system would, for practical purposes, degrade into a broadcast means of communication.

Although regions can be composed of arbitrary dimensions, for convenience, the explanations and examples throughout the remainder of this chapter will use the dimensions physical space. Figure 3 shows an example of a case in which the regions are too large in size. It presents four regions among many within a much larger virtual environment. It is easily seen that there are many entities "clumped" in region 4, but few entities in region 1, 2, or 3. If an entity, with an AOI as shown by the circle, were interested in only a small corner of region 4, it would be overwhelmed with data from entities it cares little about.

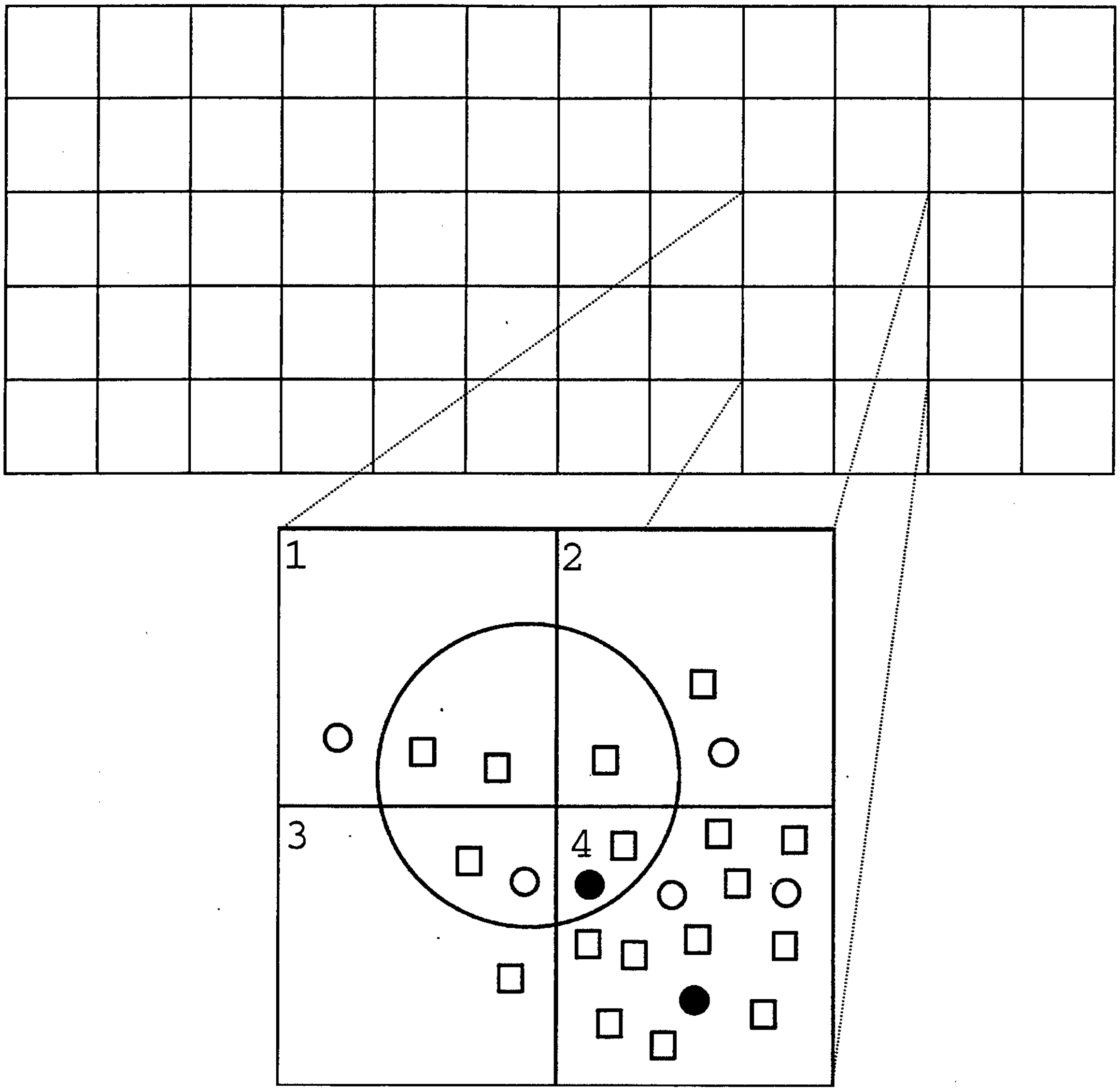


Figure 3. Example of ‘Clumping’

For the military simulation STOW-E, it was concluded that a multicast region size of 2 to 2.5 km provided significant reduction in total bandwidth usage, and that sizes less than 2 km provided only marginal additional benefit (Rak and VanHook 1996). It was also concluded that, “if the multicast grid could be dynamically re-sized and re-aligned locally, relative to the areas of highest activity, a significant reduction in total host download would be achieved."

For another military simulation (Macedonia, Zyda et al. 1995) the optimal region size was determined to be a hexagon with a radius of 4 km, this is approximately the equivalent area of a square region with a side length of 6.4 km. One possible explanation for this difference in optimal region size is that these values were determined empirically for different specific simulated military exercises. That is to say, region size is a function of the specific simulation, and a single number cannot be determined for all simulations.

One simple solution to the problem of determining region size is to use an octree to load balance these regions, and therefore the number of entities within a multicast group. If too many entities fall within one region, the octree simply subdivides it into eight regions. If too many entities leave a sub-tree of regions, eight child regions are merged into the parent region. By load balancing the multicast groups in this manner, it is impossible to encounter the clumping example described above. Figure 4 shows the same distribution of entities as in Figure 3, but with load balanced regions. The area shaded with diagonal lines represents the portion of the world that has been filtered by octree subdivision. Notice that an entity with an AOI as shown by the circle, will only receive information about 12 other entities, instead of 24 as in Figure 3.

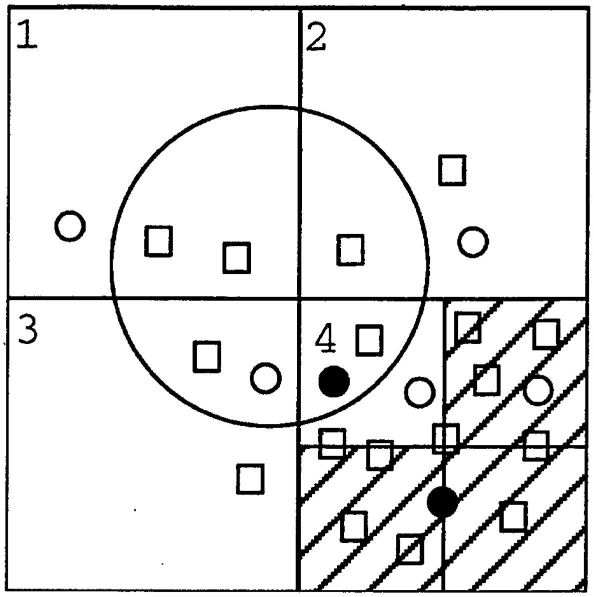


Figure 4. 'Clumping' with dynamic load balancing

However, the dynamic subdivision of the octree creates a new problem. If there is a very high density of entities, subdivision can occur to the point where entities are continuously switching regions and adding both subscription and transmission overhead when it is needed least. Take for example the case of a man standing next to an anthill covered with 10,000 ants. Because of the high density of ants, the octree will subdivide until it meets some criteria of entity density. This is exactly what the ants need, but exactly what the man does not. If the man were to move across the hill, he would go through tens, possibly hundreds, of regions with each step.

The concept of a smallest region is introduced to solve this new problem. An entity calculates a minimum size requirement that is the smallest region that it deems reasonable given its size and speed. When a region that an entity is in divides, it simply checks to see whether the new subregions are smaller than its minimum size requirement.

If so, it continues to transmit via the current region instead of switching to one of the eight new subregions.

The outcome is that entities are found throughout the octree, not only in the leaf nodes, and they are distributed not only by location, but also by a function of their size and speed. This has the added benefit for additional filtering based on size, and the ability to do efficient aggregation. Again take the example of the man and the ants. It may be that the man is running through a field, and happens to pass by the hill. If so, he may not be interested in things that are the size of ants, and as such, need not subscribe to regions small enough to hold the ants. An aggregate entity representing an aggregate view of the ants may be present in one of the larger regions, giving the man the impression that the ants are there as he passes by. But if he stops to examine the hill, he can simply subscribe to the smaller regions temporarily, to see the ants in all of their detail. It should be noted that in the real world, people do not encounter a situation between themselves and 10,000 ants. A person's brain would automatically filter such information to prevent an overload of irrelevant data. This filtering is the goal of any interest management system.

The smallest region for an entity should have a high probability of containing an entity of a given size, with a given speed, for a given amount of time. A dimensional analysis provides insight into the problem of picking the smallest region. On an intuitive basis ***M***, the minimum length of a side of a region, is expected to depend on:

1. The size of the entity, represented by some chosen reference length. Here the radius of the bounding sphere for the entity, ***R,*** is chosen.
2. Average maximum speed of an entity, ***S***.

3. Average time the entity should remain within the region, ***T***.

In light of the above, and without any *a priori* knowledge about the variation of ***M***, one can naively express M as:

$$M = K[f(R,S,T)] \tag{1}$$

where ***K*** is a constant of proportionality. Knowing that when ***S*** = *0*, ***M*** must still be at least as large as ***R***, it can be guessed that:

$$M = K[f(R) + f(S,T)] \tag{2}$$

By using dimensional analysis, and knowing that ***M*** is in units of length, it must be that *f(**R**)* and *f(**S**,**T**)* must also have units of length. ***R*** is already in units of length, so it can be said *f(**R**)* = ***R***. For *f(**S**,**T**)* to have units of length, ***S*** and ***T*** can simply be multiplied, such that *f(**S**,**T**)* = ***ST***. Putting the above back into equation 2 yields the simple equation:

$$M = K(R + ST) \tag{3}$$

Looking at equation 3, it is easy to see that ***M*** is simply a multiple of the radius of a sphere that represents the possible bounds of an entity after ***T*** seconds. ***K*** can then be thought of as a scalar to prevent the entity from leaving the region too soon. The worst case assumption can be used in which the entity travels as quickly as possible along the shortest distance toward the region boundary. For example, if ***K*** = ***1***, the only possible way for the entity to remain in the region for ***T*** seconds is for it to start in the exact center. For an entity that moves in only two dimensions and probability of 50% that in the worst case the entity fits completely within its region after the given time, ***K*** can be found using the ratio of the areas of two squares, as illustrated Figure 5.

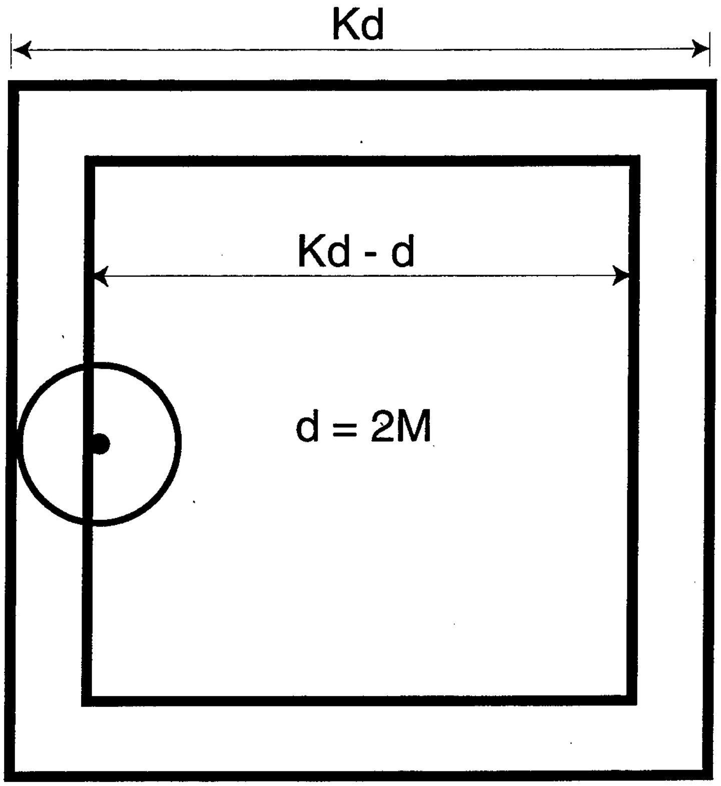


Figure 5. Area ratio of two squares

Solving for *K* yields:

$$\frac{(Kd-d)^2}{(Kd)^2}=0.50 \tag{4}$$

$$\frac{Kd-d}{Kd}=0.71$$

$$1-\frac{1}{K}=0.71$$

$$\frac{1}{K}=0.29$$

$$K=3.45$$

Similarly, for entities that move in all three dimensions and a probability of 50%, ***K*** is calculated as the ratio of the volumes of two cubes:

$$\begin{aligned} \frac{(Kd-d)^3}{(Kd)^3} &= 0.50 \\ \frac{Kd-d}{Kd} &= 0.79 \\ 1-\frac{1}{K} &= 0.79 \\ \frac{1}{K} &= 0.21 \\ K &= 4.76 \end{aligned} \qquad (5)$$

For example, if arbitrary time ***T*** of 600 seconds (10 minutes) is chosen, a stationary tank with a bounding sphere of 5-meters in diameter would only need a minimum region size of 24 meters. However, when the tank is moving at an average of 1.6 m/s (Macedonia, Zyda et al. 1995), it would need a minimum region size of 4593 meters. This region size is comparable with the average of previous region size estimates (2500 meters (Rak and VanHook 1996) and 6450 meters (Macedonia, Zyda et al. 1995) averaging 4475 meters) for military ground vehicles.

Some types of entities may have special requirements. For example an entity which makes a very loud sound, or has a very bright light may want to position itself in a larger region than calculated because it can be seen or heard from a farther distance away. To accomplish this, an entity can just scale the ***R*** value to account for the enhanced visibility. Perhaps the tank in the last example has a bright spotlight on top of it that allows it to be seen from twice the distance away than a normal tank. To accommodate this special feature, ***R*** would then be scaled from 5 to 10 meters, yielding a smallest region of 48 meters compared to 24 meters for the stationary case, and 4617 meters compared to 4593 meters for the moving case.

Because a formula for deciding the smallest region an entity can subscribe to is now known, such requirements can be placed in an entity's interest expression. For example, given the size and speed of the smallest entity that a client is interested in, the same formula can be used to decide what is the smallest region a client should subscribe to. Of course, this is just a guideline, and the choice of which regions to subscribe to is still left up to the client. For example, a client can subscribe to only very large regions which are far away, and subscribe to progressively smaller and smaller regions as they grow nearer. The client can also determine its network and CPU load and dynamically subscribe and unsubscribe to larger or smaller regions based on this load.

To address the issue of what should be considered "low" fidelity and a "low" rate, the first tier uses an algorithm called "dead reckoning," which only sends out state messages consisting of position, velocity, and time, whenever an error threshold has been exceeded. Clients receiving the state messages can accurately predict the entity's position within the error threshold by using the simple equation of motion:

$$\vec{P} = \vec{P}_0 + \vec{V}_0(t - t_0) \qquad (6)$$

Figure 6 shows an entity's actual and predicted positions over time. Packets are only sent when the error threshold is exceeded. Fidelity is a function of the error threshold, which, like regions size, should depend on an entity's size and speed. Therefore, it can now be said that the error threshold is proportional to the size of an entity's smallest region.

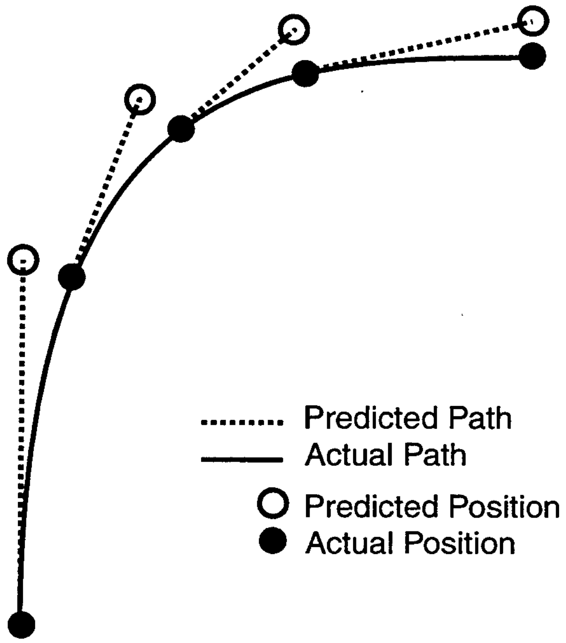


Figure 6. Dead Reckoning Algorithm

From the above statements, intuitively, this problem can be written as:

$$E = cM \tag{7}$$

The constant of proportionality ***c***, can then be thought of as a percent error with respect to the region size. For the low fidelity updates to be within 1% of the actual positions when compared to the size of the region, ***c*** would simply be 0.01. Again consider the example of the moving tank with a region size of 4593 meters. With a ***c*** value of 0.01 the error threshold would be about 46 meters. An ant on the other hand with a smallest region size of 4 meters, would have an error threshold of just 4 centimeters.

It was stated earlier that octree division is based on entity density. The reasoning behind dividing the octree is that there would be too many state packets being sent if all entities were clumped into one region. The more state packets, the more bandwidth and processing that is needed.

A packet is sent from an entity whenever the error threshold is exceeded. The worst case scenario is that after sending a state packet the entity immediately turns in the opposite direction, thereby diverging from the predicted position at a rate of ***2S,*** but this is rare. Instead it can be said that the average worse case is that the entity stops, and therefore diverges at a rate of ***S*** from the predicted position. If $\boldsymbol{P_{max}}$ is the maximum number of packets per second that can be tolerated per region, then $\boldsymbol{N_{max}}$ is the maximum number of entities that can be tolerated before the octree region is divided. Based on an entity's error threshold and speed it can be written:

$$N_{\max} = \frac{P_{\max} E}{S} \qquad (8)$$

Substituting equation 7 and 3 into equation 8 yields:

$$N_{\max} = \frac{P_{\max} cM}{S}$$
$$N_{\max} = \frac{P_{\max} cK(R+ST)}{S} \qquad (9)$$

If it is assumed that ***R*** is small when compared with ***ST***, then equation 9 can be simplified:

$$N_{\max} = \frac{P_{\max} cKST}{S}$$
$$N_{\max} = P_{\max} cKT$$

(10)

Equation 10 contains no entity dependent terms, so N_{max} is entity independent and represents the threshold at which the octree should be divided. For example if clients can handle regions with a rate of $P_{max} = 2$ packets per second, using previous values of T, c, and K, the octree would be divided at $N_{max} = 54$ entities, guaranteeing an average maximum rate of 2 packets per second per region.

The point at which to merge octree regions is then a simple ratio. To achieve at most N_{max} entities in a single octree region, the number of entities in the leaves of an octree cell must be less than N_{max}. If the extreme case is that out of the eight leaves all the entities are in a single leaf, then there exists a minimum number of packets per second P_{min} that must be maintained. The minimum number of entities in the eight adjacent leaf cells N_{min}, can now be expressed as:

$$N_{\min} = \frac{P_{\min}}{P_{\max}} N_{\max} \qquad (11)$$

For example, if the previous value of $N_{max} = 54$ and a minimum number of packets per second, $P_{min} = 1$, are used, then the number of entities at which point the regions would merge is $N_{min} = 27$.

One of the limitations of IP multicast stated in Chapter II was that joining or leaving a multicast group is not instantaneous. In fact, the latency can be on the order of one half second or more. If the first tier is implemented using IP multicast, this problem can be accounted for when deciding on an AOI. For example if an entity can move within the virtual environment at a speed of 100 meters per second, the radius of its AOI can be extended by 50 meters to account for a 0.5-second lag in the time it takes to join a group.

By adding to the AOI in this fashion it is possible that the client may receive information about more entities than is wanted. However, region-based filtering is only a first pass and the second tier will decide if indeed an entity's higher fidelity information is potentially needed.

2. Tier Two: Protocol-Independent Filtering

A client can use the information gained from the first tier filtering to limit the list of possible entity candidates to choose from, thereby limiting the amount of network and CPU resources needed for interest management. By examining only the data in the first tier, the search problem is reduced from O(*N*), to O(*M*), where *N* is the total number of entities in the virtual environment, and ***M*** is the number of entities in the octree regions an entity is interested in. Furthermore, if the first tier is implemented using multicast groups, than this initial reduction from ***N*** to ***M*** is accomplished almost entirely by the network hardware itself, the client only needs to specify which regions are of interest. Figure 7 shows the same AOI and region intersections from Figure 4, but now the entity chooses only in entities that are within the AOI. The AOI represents a protocol-independent IE and the dimensions in which this AOI is placed represents a common attribute space for all protocol-independent IEs to filter from.

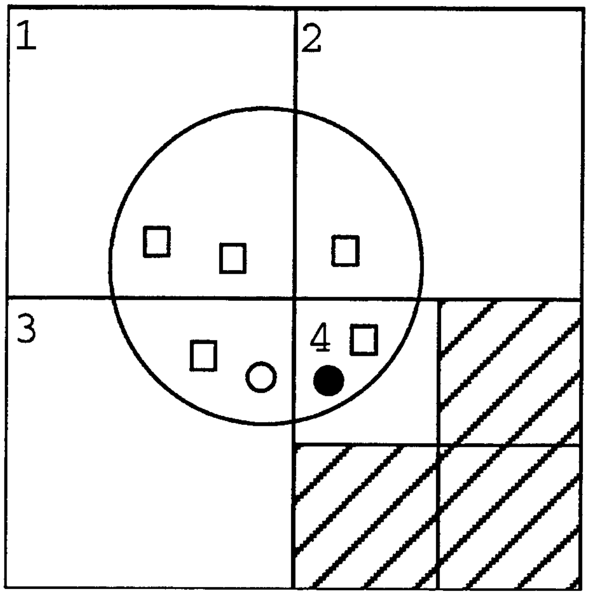


Figure 7. Load balancing with AOI filter

The manner in which data is gathered in the first tier need not be protocol-specific. In fact, it has already been demonstrated how multiple protocols can be dynamically inserted into a simulation at run time (Watsen and Zyda 1998). It has also been shown that using a multicast address per data stream is optimal for long-duration data flow applications in terms of both bandwidth and CPU usage (*Levine, Crowcroft et al. 1999). If each entity were to have its own multicast address, clients could subscribe to each other on a per entity basis. An added benefit to having a network stream per entity or protocol is that the streams can be application protocol specific, and can bypass the generic IM layer all together. The protocol specific streams can also make general assumptions to take advantage of the specifics of the entity states that it is trying to communicate. For example, if a generic protocol is trying to transmit the position of a train, it can do so by regularly sending state packets containing the train's position.

However, if a train-specific protocol is used to transmit the position of a train, it can simply send the track once, and then send the train's velocity and position whenever they change dramatically from the predicted values.

Figure 8 shows the same AOI and region intersections from Figure 4, but now the entity is interested only in entities that are within the AOI and using the circle protocol.

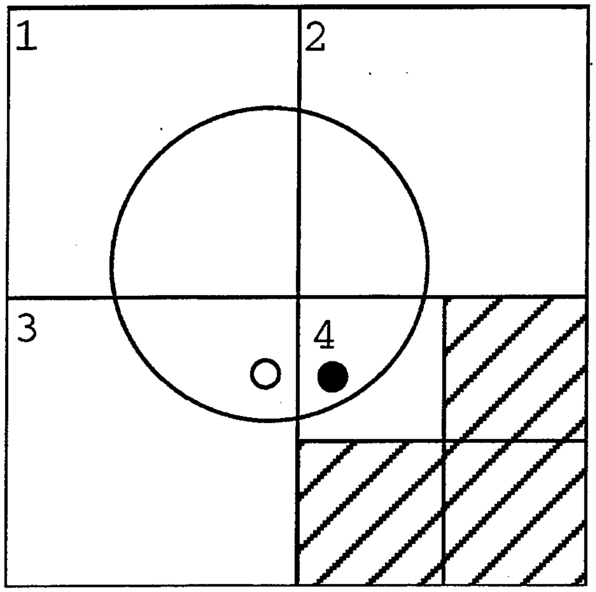


Figure 8. AOI and protocol filter

3. Tier Three: Protocol-Dependent Filtering

The first two tiers can be implemented in a protocol-independent manner, so that a simulation consisting of multiple protocols can operate without performing interest management specifically for each protocol used within a client.

This introduces a problem. If the interest management software is unaware of specific attributes of a given protocol, than these attributes cannot be contained within a client's IE and used for filtering. By adding interest management information specific to a protocol, a third tier can be created to allow an almost perfect filtering of data.

Continuing the example from Figure 8, Figure 9 again shows the same AOI and region intersections, but now the entity is interested only in entities that are circles, and have the protocol specific property of being solid in color. This yields only one entity, rather than 24 as in Figure 3.

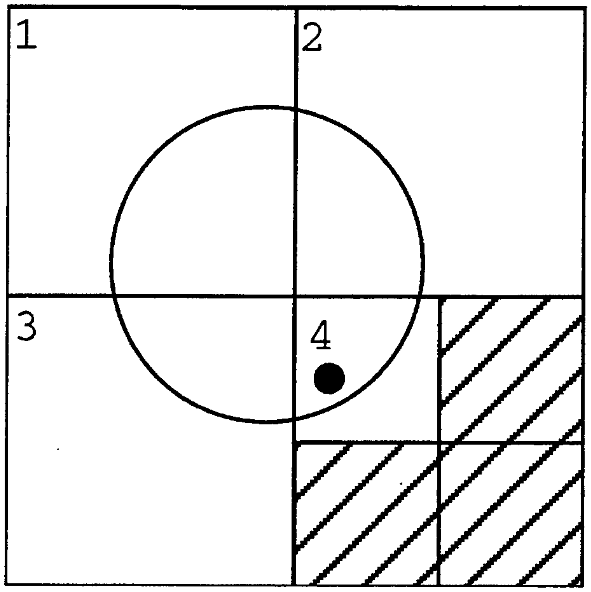


Figure 9. AOI and protocol specific filter

The second tier simply hands a packet from an entity to the correct protocol-specific filtering module, and then that filter decides how or whether to obtain information about that entity. For example, an entity might publish to multiple multicast addresses, each with a specific type of data or at a different rate. By selecting from among there groups, a client can receive exactly the data it needs.

Alternatively, to reduce the number of multicast addresses used, a protocol can aggregate entities into a fixed number of multicast groups. Although this has been shown to be less efficient in terms of both bandwidth and CPU usage (Levine, Crowcroft et al. 1999), today's routing hardware and software can only handle thousands of multicast addresses, not millions.

Figure 10 continues the example from Figure 8, but this time the entity is interested in any color circle, but with increasing fidelity as it approaches the center of the AOI.

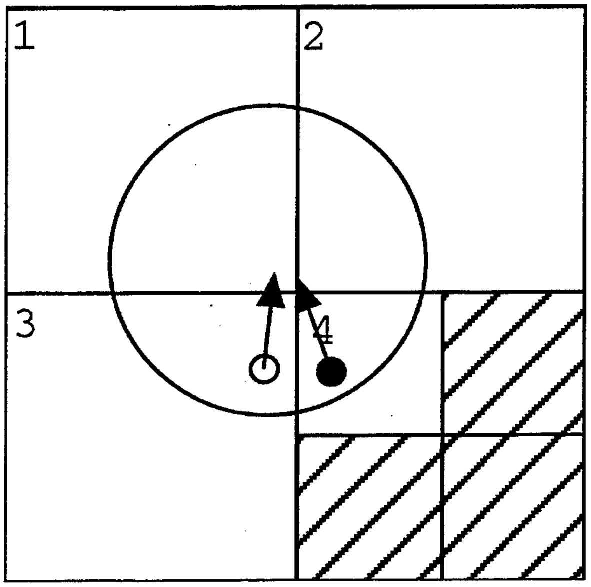


Figure 10. AOI with varying fidelity

Of course a client can be implimented so that is doe not subscribe to any individual entities, but only to the regions found in the first tier. This can be very useful for Plan View Displays, where information about many, perhaps hundreds of thousands of entities is needed, but only at a low rate, and at low fidelity.

E. SUMMARY

This chapter presented the theory and design of the Three-Tier interest management system. Specifically it outlined the overall goals for a successful interest management system, and the theory behind each of the three tiers in the system presented in this dissertation. The next chapter presents the implementation decisions and details behind the design.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

A. INTRODUCTION

This chapter presents the implementation decisions and details behind the design of the Three-Tier interest management system. Specifically it explains the rationale behind its component design choices, use of a server, and class hierarchy.

B. BAMBOO

To successfully implement a virtual environment that can never be shut down, portions of the simulation software need to be updated, removed, or added at runtime. For this reason Bamboo (Watsen and Zyda 1998) was chosen as the underlying software engineering framework for the Three-Tier interest management system.

Bamboo simply provides a language-and-platform-neutral framework to support the loading and unloading of executable code or data at runtime. Data or code is packaged into units called “modules.” A module is simply any filesystem subdirectory hierarchy, containing a file called ”.module.txt” in its root. This file contains all of the information needed to load and unload the module. This information includes the function that should be called when the module is loaded, the function to call when the module is unloaded, and dependencies regarding which other modules that should be loaded into memory before the current module can be loaded.

C. MOSTLY SERVER-LESS ARCHITECTURE

For a system to be fully scalable yet reduce transmission latency, it should only use peer-to-peer networking, and not have a server. Two problems emerge within the Three-Tier architecture without a server: octree consistency and lost entities.

If the simulation is to be fully distributed then the entities themselves must determine when to divide or merge the octree. To accomplish such decision making this one can use the concept of a master entity in each region (Macedonia, Zyda et al. 1995) or a randomized response timer (Floyd, Jacobson et al. 1997); unfortunately both of these approaches have the same flaw when using unreliable communication methods. In the case of the master entity, there is a chance that two entities could both believe that they are masters, and both could divide the octree, creating two parallel virtual environments with some entities in each. The same scenario could happen with a randomized timer algorithm; that is, more than one entity could respond at about the same time, and create parallel environments. The creation of these parallel universes would be fatal to a persistent virtual environment.

Another problem is that entities can become "lost." As entities move from region to region, they must periodically obtain information about the octree so that they may subscribe to the correct multicast address. If entities are distributed evenly throughout the octree, and all clients are aware of the twenty-six octree regions surrounding the one that they are currently in, then this task can be accomplished easily, and there would never be a gap in information. However, suppose that a client controlling a single entity is moving from region to region, when suddenly all of the entities around it decide to leave the virtual world. When the entity moves to an unexplored area, it will have no neighbors to ask for information about the octree. At this point the entity would become 'lost' leaving its current region without knowledge of what multicast addresses lay ahead.

Using a lightweight server that keeps a complete copy of the octree in memory can solve both the octree consistency problem and the lost entity problem. The server answers two basic requests. To solve the octree consistency problem, whenever a client decides to divide or merge the octree, it simply sends a reliable message to the server asking for permission. The server only grants the first client that asks permission, thereby keeping the octree consistent. Clients can still use randomized timers to delay the sending of messages to the server, to avoid overloading it with useless redundant messages.

Because the server always has a consistent copy of the octree in memory, it can handle search queries about the octree. For example an entity can state its location and radius of interest and ask for a list of octree regions that overlap this interest radius. This way an entity always has a reliable source of octree information.

This is not to say that an entity should rely on the server on a regular basis. Everything besides octree consistency can remain server-less, and the server should be used only as a fail-safe in case a client cannot get the information it needs from other clients.

The actual server implementation is a very small program, written in less than 500 lines of C++ code. The only information stored in the server is the octree structure and a multicast address for each region. Each node in the octree requires only 12 bytes of storage. If entities were only present in the leaves of the octree and each node could hold

40 entities, then the server could store an octree large enough for 5 billion entities within a 32bit memory address-space. For a virtual environment that is persistent, a fault-tolerant mechanism to distribute the workload among many servers must be created, although it does not exist in this implementation.

D. CLIENT BASE CLASS MODULE

The main core of the software architecture is built using a Bamboo module containing a set of C++ base classes. There are five key base classes: IMBaseClient, IMBaseEntity, IMBaseInEntity, IMBaseOutEntity, and IMBaseProtocol. Virtual environment developers need not concern themselves with the internal workings of the Three-Tier architecture, but only need to derive from these five classes to use or extend its functionality. Figure 11 presents a simplified UML diagram of these five classes. Interested programmers are urged to read Appendix A, which contains the source code for these five main base classes.

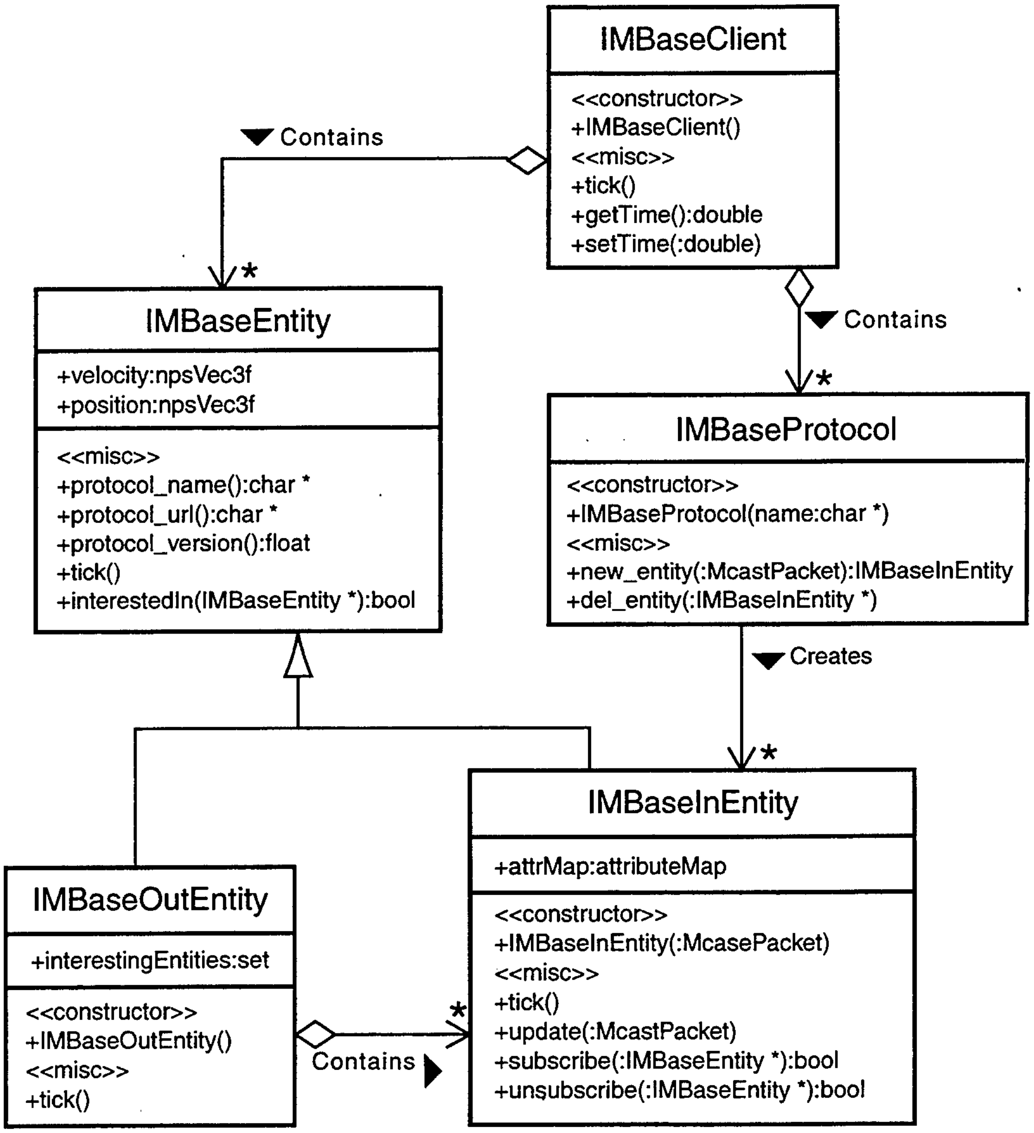


Figure 11. Simplified UML class hierarchy diagram of the five key base classes

1. IMBaseClient

The main class of the five base classes is the IMBaseClient class. It is responsible for subscribing and unsubscribing to multicast addresses, processing and tier two filtering of entities, maintaining a local partial copy of the octree, and managing the loading and unloading of protocol modules. Developers of new clients only need to derive from the IMBaseClient class, and implement client specific functionality. Examples of derived clients include stealth viewers, flight simulators, and automated entity control. Each client has two main threads of execution: a tick thread and a network thread.

The tick thread is responsible for updating the state of the client for each simulation "tick" or time unit. Specifically it updates its components in the following order:

1. *Delay Respond Queue*. The delay respond queue is a time-ordered queue that, when used in conjunction with a random timer, allows a random delay to be introduced before a response is issued for octree query.
2. *Delay Send Queue.* The delay send queue is also a time-ordered queue that allows the sending of a multicast packet to be delayed. This is used to send duplicate packets to a multicast address to allow from some packet loss as described in Chapter II.
3. *Octree Recheck Queues*. These time-ordered queues are used to check the octree against density constraints after a certain period of time. This helps

prevent octree subdivision oscillation. By placing a random delay in the queue, it can also prevent server overload.

4. *Simulation Time.* The simulation clock is incremented either by the real-time clock, or a user defined clock. All entity tick functions use this value as the universal simulation time.
5. *Interesting Entities List.* Each local entity's 'interesting entities' list is computed by comparing its location with every other entity that the client currently knows about, that is every entity which falls within the interesting octree regions. If the entity falls within the entity's second tier thresholds, it is checked against its "interestedIn" function, which is described below.
6. *Entity Ticks.* Every local and remote entity's "tick" function is called.

The tick thread must be allocated and managed by an outer application, which is responsible for deciding the simulation tick rate. There are also hooks to control simulation time, to allow for non-real-time distributed simulations.

The network thread is allocated by the baseClient itself and is responsible for receiving and processing incoming messages from the network. There are six types of messages that can originate from an octree multicast address: MOVED, PINGING, MERGE, SPLIT, QUERY, and RESPOND. When a client receives a PINGING message from an entity within a region, all local entities within that region are instructed to then send updated state messages. When a client receives a MERGE or SPLIT message, the client updates its local copy of the octree to reflect the octree structure change and then updates the effected local entities. The QUERY and RESPOND message are used to maintain the local copy of the octree without having to contact the server. Instead a random client listening to the region handles the QUERY message, and responds with a RESPOND message. The bulk of the messages that a client receives from the octree regions are MOVED messages. These represent state updates from entities within the region. Figure 12 shows a flowchart representing the path each MOVED message must take and the time complexity of each step, where ***P*** is number of application protocols known to the client, ***R*** is the number of entities in the octree regions the client is currently interested in, ***Er*** is the number of entities within a specific region, and ***Eq*** is the number of regions within the recheck queues. It should be noted that because the values of ***P***, ***Er***, and ***Eq*** are normally all small compared to ***R***, worst case this is a O(lg ***R***).

Entities express interest through interest expressions. The current client implements interest expressions using an attribute value associative array, evaluated via compiled code within the "interestedIn" function of the IMBaseOutEntity class described below.

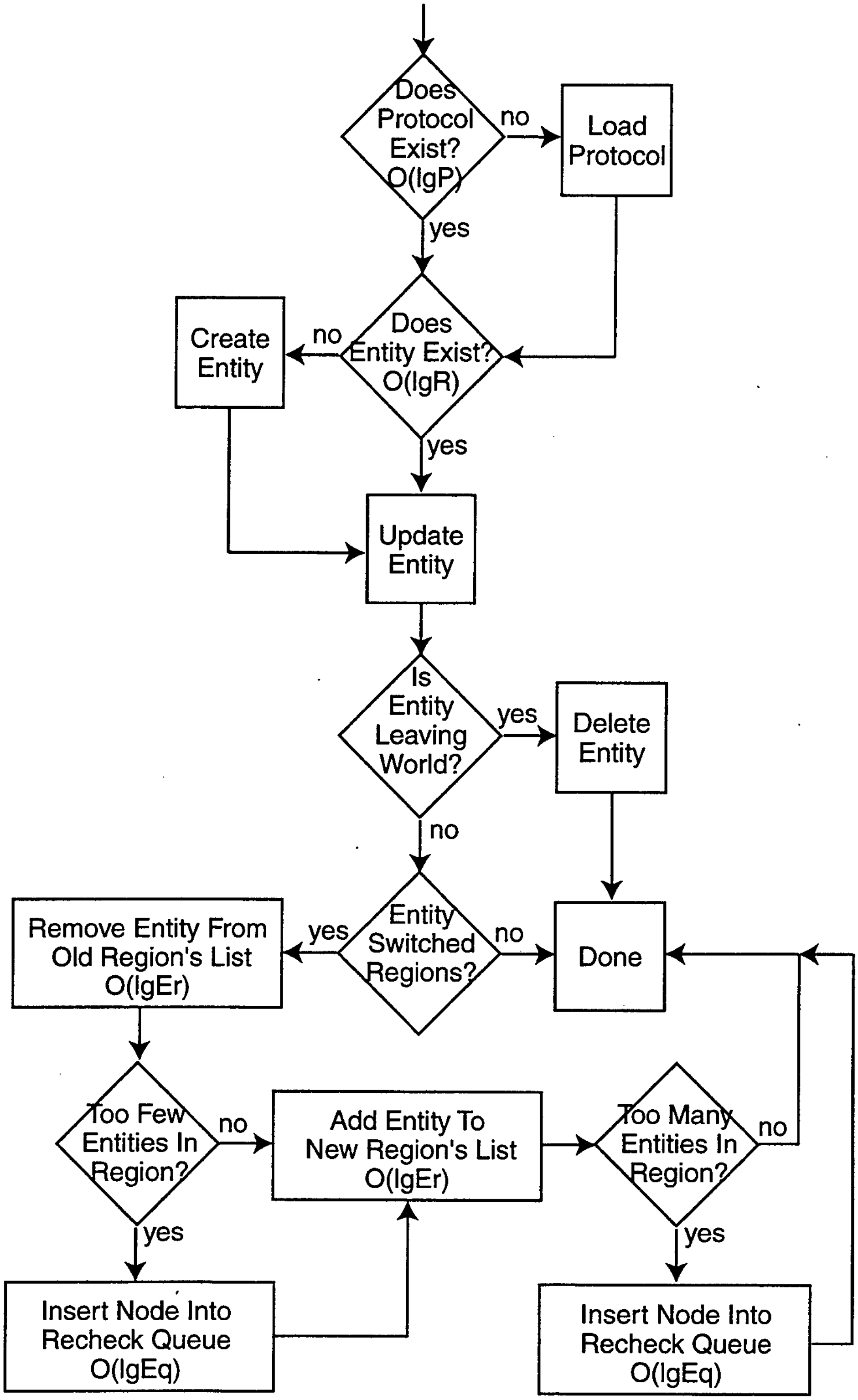

Does Protocol Exist? O(lgP)
no
Load Protocol
yes
Create Entity
no
Does Entity Exist? O(lgR)
yes
Update Entity
Is Entity Leaving World?
yes
Delete Entity
no
Remove Entity From Old Region's List O(lgEr)
yes
Entity Switched Regions?
no
Done
Too Few Entities In Region?
no
Add Entity To New Region's List O(lgEr)
Too Many Entities In Region?
no
yes
yes
Insert Node Into Recheck Queue O(lgEq)
Insert Node Into Recheck Queue O(lgEq)

Figure 12. Flowchart illustrating the path each MOVED message follows

2. IMBaseEntity

The IMBaseInEntity and IMBaseOutEntity classes are used as protocol independent handles to entities. An IMBaseInEntity represents an inbound entity reflected from the network, it contains all the code and data necessary to represent itself on the local client. An IMBaseOutEntity represents an outbound entity controlled by the local client. The IMBaseClient maintains a list of interesting entities for each outbound entity based on interest expressions contained within each IMBaseOutEntity. Classes derived from the IMBaseInEntity and IMBaseOutEntity classes also know how to communicate directly among themselves, providing a protocol dependent communication channel when necessary. An example of a derived entity is a human avatar. Using the base methods a client can learn about the human entity. The human entity modules can then communicate between themselves, efficiently relaying entity specific information such as joint angles and eye gaze direction.

To implement a derived entity's functionality, a developer needs to only implement the "tick" and "interestedIn" functions. By default an entity's "tick" function is called once each time the base client it "ticked." For an inbound entity, a standard approach would be to dead reckon and smooth entity specific data received over the network from its outbound counterpart, then call the super class' "tick" function. For example, if this entity represents a human avatar, the tick function dead reckons joint angles based on angular velocity. The super class IMBaseInEntity "tick" function then calculates the current predicted position based on dead reckoning parameters and, upon receiving a new update, smoothes position values by interpolating between old and new positions.

For an outbound entity, a standard approach is to look at other interesting entities contained within the list generated by the client, update the entity's position as well as other fields within the entity, then call the super class' "tick" function. The inherited IMBaseOutEntity "tick" function automatically calculates dead reckoning predicted error and sends new octree packets when error thresholds are exceeded. It also determines the entities' primary region and interesting regions based on octree intersection.

As described above from the point of view of the client, an outbound entity's "interestedIn" function is called during each simulation tick for every other entity that the system knows about. This simple function implements the third tier filtering. Using either compiled-in prior knowledge of other entity types, or by using the named attribute map described above, an outbound entity decides which entities are of interest to it. Continuing our example from above, an outbound fish entity may only be interested in other fish that are of its same species. If the developer wants to make the simulation more complicated, he/she could use the attribute map to express that the fish is interested in all entities that are larger in size than it, because these entities may hurt the fish. The developer could then extend it further by expressing that the fish is also interested in other fish that are smaller that it, so that it may eat them. From these simple expressions, the "interesting entities" list is formed. It is then up to the outbound entity's "tick" function to do something intelligent with this list.

Interested programmers are urged to read Appendix B, which contains the source code for the entities used in the experiment outlined in the next chapter.

3. IMBaseProtocol

The IMBaseProtocol class is a base from which all protocols can derive. It simply knows how to construct entities that use the protocol. Developers of new protocols only need to create a new protocol module. Protocol modules are Bamboo modules that contain an initialization function and three classes that are derivations of IMBaseProtocol, IMBaseInEntity, and IMBaseOutEntity. The derived IMBaseInEntity and IMBaseOutEntity classes are the protocol specific versions of the entities, which know how to communicate with each other.

The initialization function is designated as the Bamboo module initialization function, and its only requirement is to create a single instance of the derived IMBaseProtocol module. Developers can opt to package many different versions of an entity class within the module and instantiate them based on some criteria. For example, if a graphics module is already being used within the currently running system, the protocol module should instantiate a entity that knows how to draw itself, whereas if there is no graphics currently loaded within the system, a simpler, lighter-weight version can be instantiated instead.

The rationale behind using protocol modules is that they can define new ways of communicating between their entities. For example a derived protocol class can create its own socket, allowing it to send and receive its own packet format. By doing so it can optimize its transmissions specifically for its entities.

Interested programmers are again urged to read Appendix B, which contains the source code for the protocol module used in the experiment outlined in the next chapter.

E. SUMMARY

This chapter presented the implementation decisions and details behind the design of the Three-Tier interest management system. The next chapter presents the experimental design and results of measurements comparing the Three-Tier interest management system against systems using broadcast based and region-based filtering.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENTATION AND MEASUREMENT

A. INTRODUCTION

This chapter presents the experimental design and results of measurements comparing the Three-Tier interest management system against systems using broadcast-based and region-based filtering.

B. MEASURES OF EFFECTIVENESS

For the Three-Tier system to be effective, it must meet the requirements listed in Chapter I. Specifically, Packets Per Second, Bandwidth, and CPU Time must depend only on the number of entities a client is interested in at any one time. Thus each experiment records:

1. *Packets Per Second*: The average number of packets received and processed per second at each client.
2. *Bandwidth*: The average number of bytes received and processed per second at each client.
3. *CPU Time*: The total number of seconds used to complete one simulation cycle for each client.
4. *Interesting Entities*: The number of entities of interest at a given moment for each client.

5. *Application Metric*: How well does the application "work," using application specific criteria.

The experiments also seek to validate the set of formulas from Chapter III. Specifically, they show that the formulas accurately predict the maximum number of packets per second for a given region, as well as the average amount of time a given entity remains within one region. To do this, the following additional measurements are taken:

6. *Packets Per Second Per Region*: The number of packets that are received and processed, recorded on a per region basis at each client.
7. *Time Per Region Per Entity*: The amount of time that each entity spends within a region.

Also, because this work relies on networking technology, the status of the loaded network is recorded during each experimental run. It should be noted that in theory the performance of the architecture being tested should be independent of these measurements, but they are recorded in order to place this work within its networking context. Specifically the metrics are:

8. *Total Segment Packets Per Second*: The number of packets on a local LAN segment.

9. *Packet Latency*: The number of milliseconds it takes to send a packet from one computer to another.
10. *Packet Jitter*: The variation in packet latency.
11. *Time to Join*: The number of milliseconds it takes to receive a packet from a multicast group after joining a that group.

C. EXPERIMENTAL DESIGN

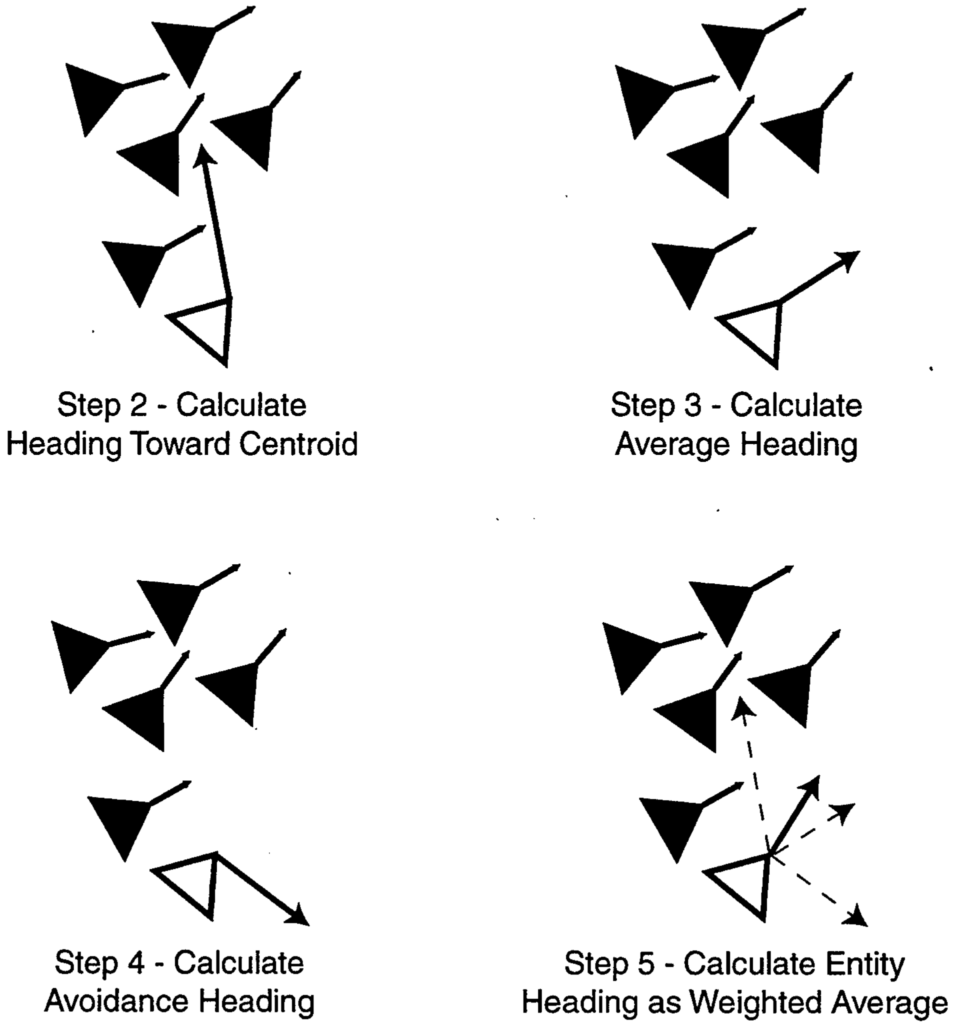
The test application scenario used for this experiment was entity flocking (Reynolds 1987). Flocking is CPU intensive for large numbers of entities, and is typically implemented as a O(N) problem to compute the next heading for a given entity, which ordinarily makes the problem O(N^2) for the entire system during each simulation update cycle. For each iteration, the algorithm works as follows:

1. Each entity calculates its distance from all other entities, and then selects entities that fall within some pre-defined radius.
2. The heading toward the centroid of the selected entities is calculated.
3. The average heading of the selected entities is calculated
4. An average avoidance heading is calculated for all entities within a smaller pre-defined radius.
5. A new entity heading is computed from the weighted average of the three headings above.

6. If the new entity heading diverges too far from the current entity heading, it is clipped based on a maximum turn rate. This prevents the entity from instantaneously changing direction.

In the implementation used within this experiment, step one is replaced with interest management. Steps 2 through 6 are performed using only the "interesting entities" list generated for each out-bound entity by the client. Because of interest management, for each entity we have reduced the problem to O($\boldsymbol{I} + \boldsymbol{M}$) per entity, where $\boldsymbol{M}$ is the number of interesting entities, and $\boldsymbol{I}$ is the cost of interest management. However it can be seen from Chapter IV that in the Three-Tiered system $\boldsymbol{I}$ depends only on $\boldsymbol{M}$ and lg$\boldsymbol{R}$, so it can be said that this algorithm is implemented in O($\boldsymbol{M}$) per entity. Figure 13 presents a diagram view of steps 2 through 5 of the flocking algorithm.

The flocking algorithm is implemented within the "tick" function of a derived outbound entity class, a fish class. This new class also has a protocol spacific attribute called "species" that represents the species of the fish entity. Each fish is only interested in schooling with fish of its own species. All three of these entity species are then run with three different base clients. Each base client filters using a different filtering scheme: broadcast-based, region-based, and Three-Tier-based filtering.



Step 2 - Calculate Heading Toward Centroid

Step 3 - Calculate Average Heading

Step 4 - Calculate Avoidance Heading

Step 5 - Calculate Entity Heading as Weighted Average

Figure 13. diagram view of steps 2 through 5 of the flocking algorithm

1. *Broadcast-based filtering client*

The broadcast based client uses DIS-like packets with a 5-second keep-alive heartbeat and a 1m error threshold, with no angular thresholds. The packets contain various state and dead-reckoning information. The resulting simulation is very similar to DIS, but with only entity state PDUs.

2. *Region-based filtering client*

The region-based client uses the same DIS-like packets as above, again with a 5-second keep-alive heartbeat, 1m error threshold, and no angular thresholds. The size of the cube shaped regions is the average of previous research estimates (2500 meters (Rak and VanHook 1996) and 6450 meters (Macedonia, Zyda et al. 1995)) which is 4475.

3. *Three-Tier-based filtering client*

The Three-Tier based client is a client using the implementation outlined in Chapter IV. The octree thresholds are determined by the formulas presented in Chapter III, using the constants of ***K=3.45***, ***c=0.01***, and ***t=300s***. Each entity has its own multicast address, and transmits the same DIS-like packets with the same 1m error thresholds, no angular thresholds, and 5-second keep-alive heartbeat.

The broadcast and region based systems were chosen for comparison because they are most competitive to the system outlined in this dissertation. Server based filtering was not used because there are clear scalability issues with systems which use server based filtering, as well as the fact that the system outline in this dissertation does not use a server for filtering information, so it does not make a good comparison. Sender based filtering was not used for comparison because it also has clear scalability issues.

The experimental design is based on a three-by-three factorial design using an analysis of variance (ANOVA) with independent variables being architecture and total entity count, this is illustrated in Table 2. Each system was run for ten minutes, a total of nine times, three times at three different entity totals: 900 entities, 1800 entities, and 2700 entities. Using an ANOVA, the ten-minute time period was determined to be more than statistically adequate during trial runs of the experiment. For each entity count, each system is run three times, each on a different day, to allow for any anomalies in the network. ANOVAs are used to show that the dependent measures for each system are significantly different. Dependent measures include average CPU time, packets per second, and number of interesting entities. These are measured at one-second intervals for a total of 1800 samples per measure per entity count per system. Because the packets used in this experiment are of fixed size, bandwidth is always equal to Packets Per Second multiplied by Packet size, and therefore does not need to be recorded. Instead of using CPU time directly, the reciprocal is used; simulation updates per second. Packets per second per region and the times of region switches were also recorded for the Three-Tier system.

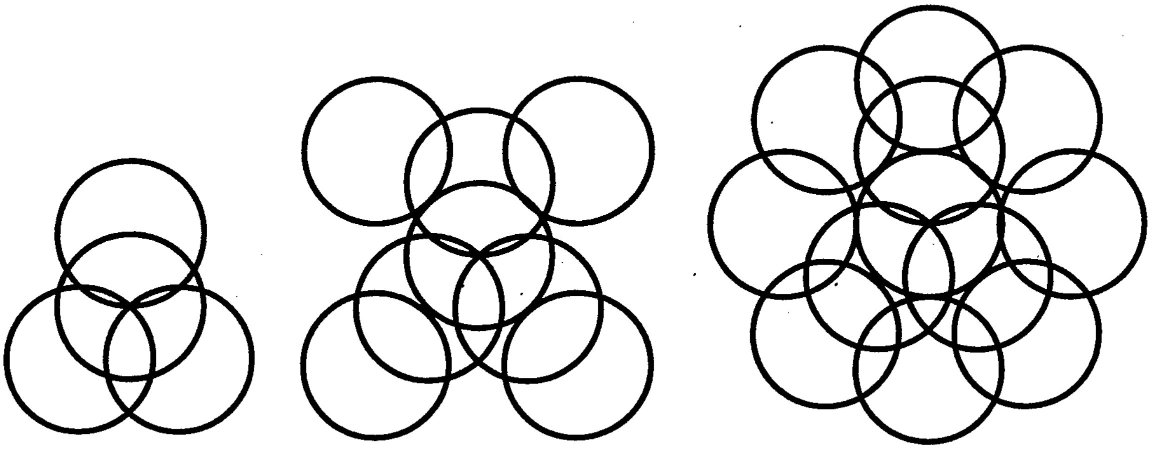
	900	**1800**	**2700**
Broadcast	1800	1800	1800
Region	1800	1800	1800
Three-Tier	1800	1800	1800

Table 2. Three-by-Three Factorial Design

Data from both broadcast and region-based filtering is compared pair-wise with the Three-Tier system using an analysis of variance. The experiments show that the Three-Tier system is significantly less dependent on the total number of entities than both broadcast and region based filtering.

The measurement of packets per second per region of the Three-Tier system is used to show that the formulas from Chapter III accurately predict bandwidth requirements. Time spent per region per entity is used to show that the formulas from Chapter III accurately predict the amount of region switching overhead per entity.

Entities are pseudo-randomly placed in groups of 225 according to Figure 14. Each group of entities is managed by a single client, running on an SGI Octane with a minimum of 256MB of memory and a 250Mhz R10000 processor. Between five and thirteen Octanes are used in each run, depending on total entity count. In each run, one Octane is designated as a data-recording client that managed only a single entity within the simulation. This client was responsible for recording all experimental metrics. The client software's tick thread is executed as often as possible, so each computer was completely utilized.



900 Entities
(4 groups of 225)

1800 Entities
(8 groups of 225)

2700 Entities
(12 groups of 225)

Figure 14. Entity distribution for experimental runs

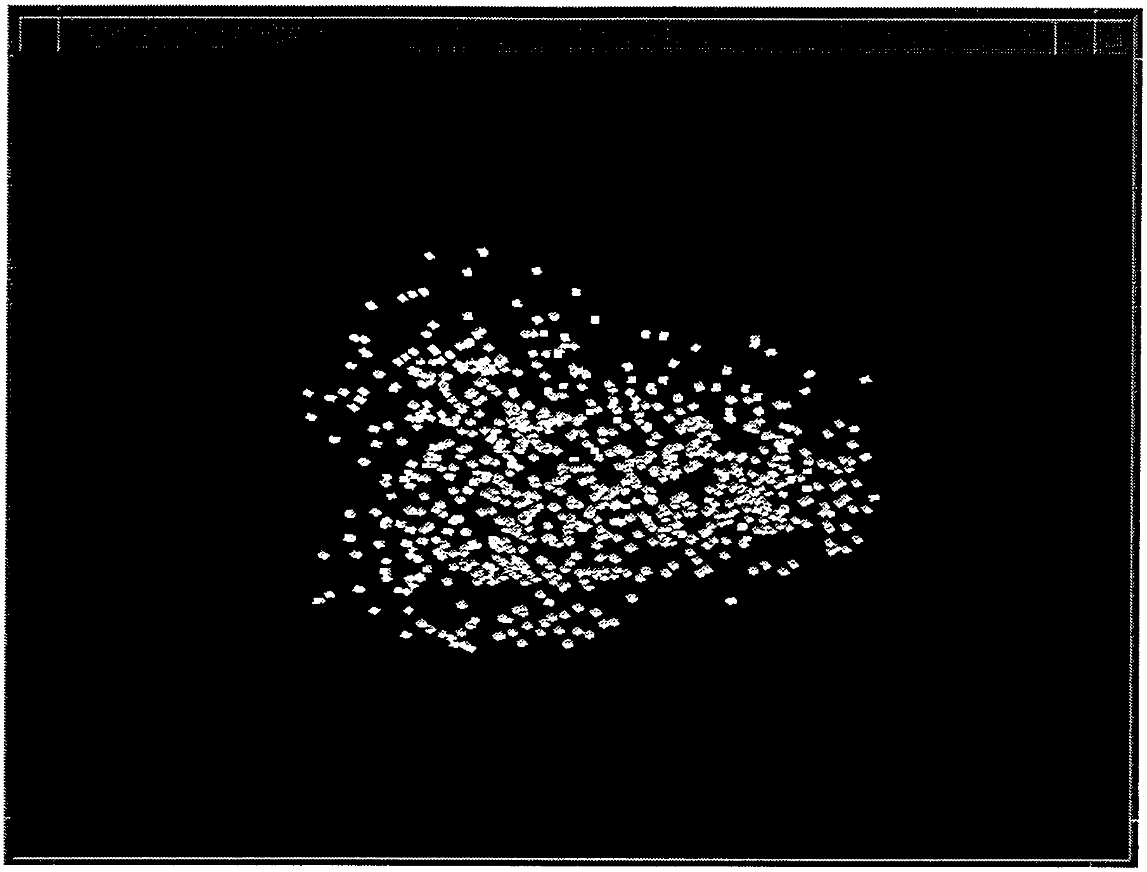


Figure 15. Experiment after 10 seconds with 900 entities

To make this experiment as fair as possible, it is set up such that the octree in the Three-Tier system does not subdivide. The resulting octree regions used are the same size and orientation as the region based system. Because of this the Three-Tier system will not scale completely dependent on the number of entities a client in interested in at any time. However because the number of regions remains fixed, this experiment can then more easliy be used to determine octree overhead within the system.

A commercial LAN-analyzer program called Observer (Network Instruments 1998), running on a laptop, is used for recording the measurements of total LAN segment packets per second, and bandwidth. Packet latency is measured using the Unix `ping` command and because packet jitter is based on this latency, it is not recorded. Because the experiment is conducted on a LAN, time to join should be a constant, so it is not recorded. This data is used to show the effect the network has on the three systems, and vice versa.

To summarize, the goal of this experiment is to show that the Three-Tier system is dependent on the number of entities a client is interested in at any one time, and much less dependent on the total number of entities than either broadcast or region based filtering. It also shows that the formulae from Chapter III accurately predict the amount of region switching overhead per entity as well as the bandwidth requirements.

These goals translate into the following hypotheses for this experiment:

H_1: A client within a virtual environment using the Three-Tier Interest Management System scales, in terms of packets per second, dependent on the number of entities that client is interested in at any one time.

H_2: A client within a virtual environment using the Three-Tier Interest Management System has a lower dependency on the total number of entities within the environment, in terms of packets per second, than that of virtual environments using either the broadcast or region based systems.

H$_3$: A client within a virtual environment using the Three-Tier Interest Management System scales, in terms of CPU time, dependent on the number of entities that client is interested in at any one time.

H$_4$: A client within a virtual environment using the Three-Tier Interest Management System has a lower dependency on the total number of entities within the environment, in terms of CPU time, than that of virtual environments using either the broadcast or region based systems.

H$_5$: The formulae presented in Chapter III work as expected. Specifically $N_{\max} = P_{\max} cKT$ accurately predicts the maximum number of packets per second for a region based on the number of entities, and $M = K(R + ST)$ predicts the average amount of time a given entity remains within one region based on region size.

H$_6$: An application within a virtual environment using the Three-Tier Interest Management System works as well as the same application within the broadcast or region based systems, as determined by application-specific metrics, in this case "flock" size.

H$_7$: A virtual environment using the Three-Tier Interest Management System has approximately the same impact on the network in terms of packets per second, bandwidth, and latency, as broadcast or regions based systems.

D. DATA AND ANALYSIS

This section presents an analysis of the results obtained from the experiment outlined in the previous section.

1. Packets per second

Figure 16 presents a graph of the average number of packets per second (PPS) received by a client versus the total number of entities within the simulation for each of the three systems. When this portion of the data was first analyzed it seemed as though the client was not receiving nearly as many packets as expected for broadcast and region based systems.

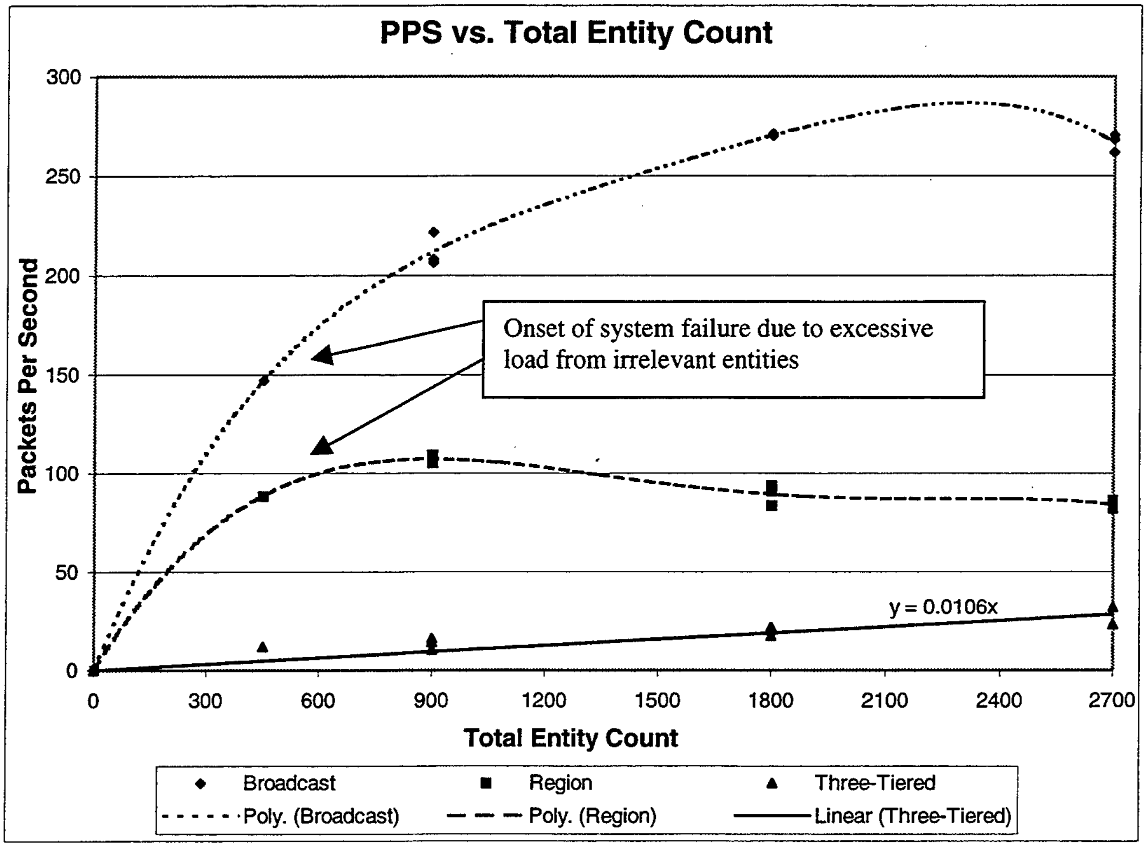


Figure 16. Packets Per Second vs. Total Entity Count

In fact, if one were to look at the data for just 900 through 2700 entities as outlined in the experimental design, one could almost come to the conclusion that region based filtering does better as the simulation grows, clearly that is not the case. By adding two more data points to the graph, one at 0 entities, and one at 450 entities, the reason for this behavior becomes clear. It was expected that broadcast and region based filtering would fail under heavy load, but it appears that they failed much earlier than expected. By again looking at Figure 16, we can see that broadcast filtering begins to taper off at about 300 entities. It is known that a properly working broadcast client should have received all packets broadcast in the simulation, so if it received 100 packets per second at 300 entities, it should have received 300 packets per second at 900 entities. However, at 900 entities the figure shows broadcast filtering at only approximately 210 packets per second, so at this point broadcast is receiving 50% fewer packets then it should have. This is due to CPU load, and will be explained in more detail in the next subsection.

Region based filtering has a similar problem, although because it uses network filtering, its failure point cannot easily be obtained from this graph. From network data presented later in subsection 5 of this section it can be seen that its failure point is near 900 entities, although it does not nearly degrade as much as the broadcast based system. Although not evident in Figure 16, the Three-Tier system also began to show reduced packet counts at high entity counts, but to a much lesser degree and only as the entity total approached 2700 entities.

Even with these reduced packet counts, the Three-Tier system performed much better than region based and broadcast based filtering. By plotting a fitted line, and calculating its slope it can be seen that packets per second grew at only 0.0106 packets per entity in the total system.

This result satisfies hypothesis H_2, that is: A client within a virtual environment using the Three-Tier Interest Management System has a lower dependency on the total number of entities within the environment, in terms of packets per second, than that of virtual environments using either the broadcast or region based systems. Furthermore, these results are confirmed by the statistics of F(2, 8636.537) with all three comparisons being significant having a $P < 0.0001$.

The goal of the Three-Tier system is to grow dependent only on the number of entities a client is interested in. To calculate the system's actual scalability, Figure 17 presents a graph of the average number of packets per second per entity of interest, versus the total number of entities within the simulation. A fitted line is plotted for the Three-Tier system, but because data from broadcast and region based systems cannot be directly correlated with the number of entities of interest, only the raw data points for those systems are shown.

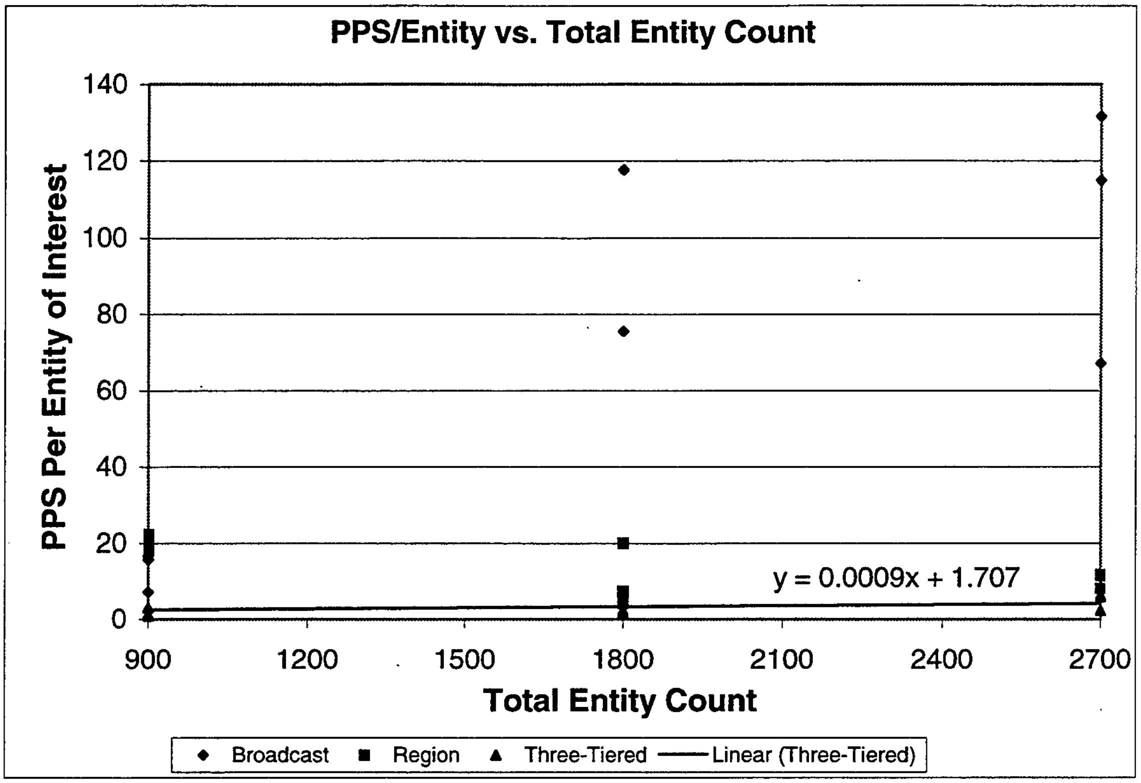


Figure 17. Packets Per Second Per Entity of Interest vs. Total Entity Count

To meet the difficult requirements of a perfectly scalable system, the slope of the line in the above figure should be 0, and its y-intercept should be the average number of packets per second each entity transmits. Although the Three-Tier system comes close, it can be seen from its slope that it has an overhead associated with total entity count. This overhead is due to the fact the client must receive information from the octree regions in order to select the entities that are of interest. However, it is expected that as the total number of entities grows the octree would subdivide and the slope of the line would approach 0. This satisfies hypothesis H_1, that is: a client within a virtual environment using the Three-Tier Interest Management System scales, in terms of packets per second, dependent on the number of entities that client is interested in at any one time.

At a total region entity count of 10,000 this overhead of 0.0009 translates to 9 packets per second, per entity of interest. In other words, in an environment with billions of entities, if a client was interested in octree regions containing 10,000 entities, and of those was really interested in only 100 of them, the client would require a network connection able to handle 900 packets per second. Using DIS size packets, this translates to a bandwidth requirement of 1 Mbps; this amount is easily handled by 10-Base-T, T1, cable modems, or even DSL, which many businesses and households already have.

2. CPU Time

Figure 18 presents a graph of the average number of CPU seconds per update versus the total number of entities within the simulation for each of the three systems tested. Again because of the packets per second anomaly described in the previous subsection, two extra data points were added to the chart to help explain, one at 0 and another at 450 entities.

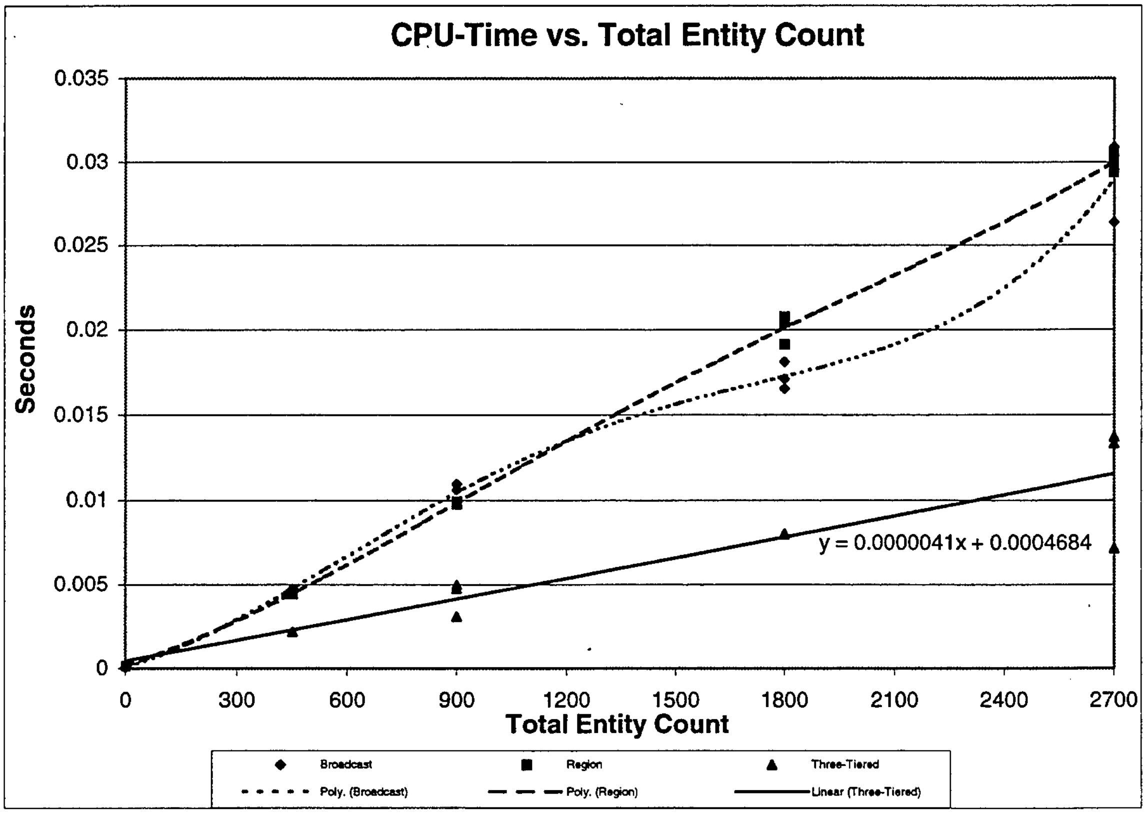


Figure 18. CPU-Time vs. Total Entity Count

Although the Three-Tier and region based systems have a fairly linear response to total entity count, the broadcast based system had a very non-linear response. This can be explained in the following manner.

Each of the clients running in the experiment controlled 225 entities, with the exception of the one client that recorded the data, which controlled only one entity. Based on the data presented above earlier in Figure 16, it is estimated that the entities used within this simulation on average each transmited one packet about every three seconds. A client would have to update at least once every three seconds to keep up with this packet rate. It can be seen from the data above, that for the Three-Tier system at 2700 total entities, it took at most approximately 0.013 seconds to update the client. If you multiply this by the 225 entities the other clients controlled you obtain an estimate of 3 seconds update time for the other clients.

Although this is barely acceptable, the other clients required more CPU time because they receive many more packets per second. This result is even more significant because even though the total number of packets on the network was higher for the Three-Tier system, CPU-time was lower. This leads one to believe that although filtering multicast packets from the network in software is more expensive in terms of CPU-time than filtering in hardware, receiving and processing these packets is an even more expensive operation. Because of this overhead for processing each packet, the other systems reach the 0.013-second mark at closer to 1000 entities. Once the broadcast based system passes this point a non-linear behavior emerges. This behavior is due to the fact that CPU time is closely related to the number of packets per second, which for the broadcast based system, is directly related to the total number of entities. As CPU would rise past the 0.013-second mark, a given client would produce fewer and fewer packets per second per entity it controls, so the CPU time required to receive all of these packets did not increase as dramatically. The region based system performed much more linearly because the number of packets a given client receives is not directly related to the total number of packets transmitted.

Once again, as with the analysis of packet rate in the previous subsection, even if these reduced CPU-times for broadcast and region based systems were accurate, the Three-Tier system performed much better. The fitted line presented in the graph only has a slope of 4.1 microseconds per entity in the total system, compared to approximately 11.2 microseconds for region based system.

This result satisfies the hypothesis H_4, that is: A client within a virtual environment using the Three-Tier Interest Management System has a lower dependency on the total number of entities within the environment, in terms of CPU time, than that of virtual environments using either the broadcast or region based systems.

Furthermore, these results are backed by the statistics of F(2, 951.939) with the comparisons between the Three-Tier system and both broadcast and region based systems being significant having $P < 0.0001$. The comparison between the broadcast and region based systems had a $P = 0.4607$, which is not significant.

Although the Three-Tier system performed much better, it is interesting to look in detail at the far-left side of the graph, shown as an enlargement in Figure 19. It can be seen from the figure below, that the Three-Tier system is not always the best choice for small-scale virtual environments. In fact, it seems to only perform better than region based systems at entity counts above 40. Furthermore, at least for this scenario, a broadcast based system would have been a slightly better choice than a region-based system until the virtual environment reached a total of 250 entities.

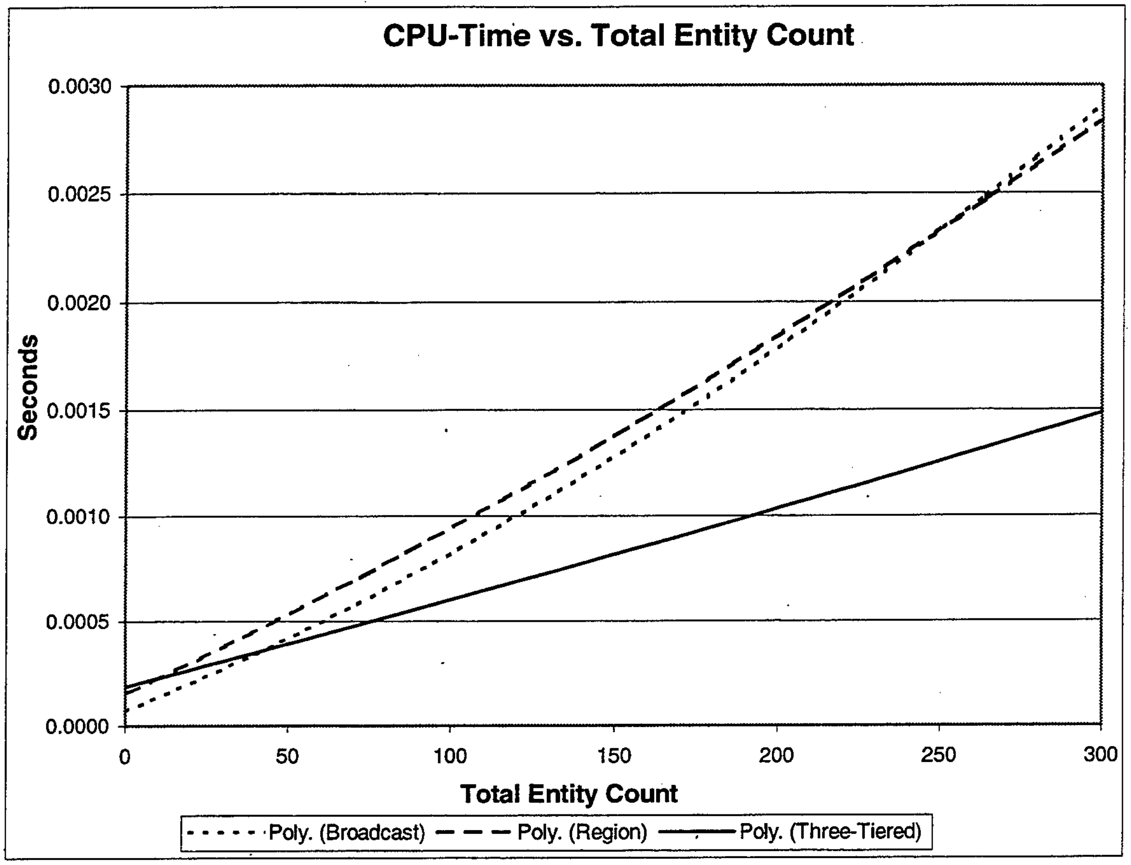


Figure 19. Enlargement of CPU-Time vs. Total Entity Count

But recall again, the goal of the Three-Tier system is to only depend on the number of entities that a given client is interested in. To analyze the Three-Tier system's scalability, Figure 20 presents a graph of the average CPU-time per entity of interest, versus the total number of entities within the environment. A fitted line is plotted for the

Three-Tier system, but because data from broadcast and region based systems cannot be directly correlated with the number of entities of interest, only the raw data points for those systems are shown.

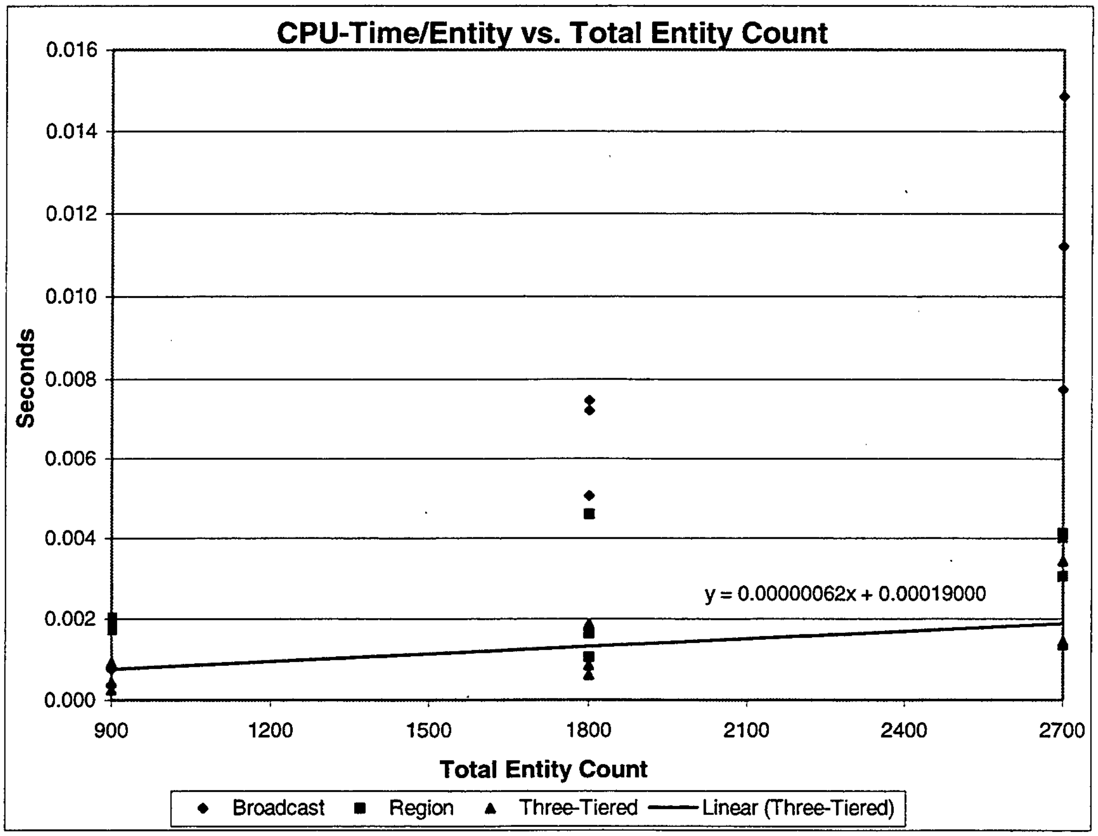


Figure 20. CPU-Time Per Entity of Interest vs. Total Entity Count

If the Three-Tier system were perfectly scalable, the fitted line above would have a slope of 0. However because this slope is small when compared to the y-intercept, it can be said that this satisfies hypothesis H_3, that is: a client within a virtual environment using the Three-Tier Interest Management System scales, in terms of CPU time, dependent on the number of entities that client is interested in at any one time.

Although the slope presented is small, it does represent the overhead associated with dead reckoning entities that are within the regions of interest, but are not of interest themselves. To put this overhead in context, we continue the example from the previous subsection. If a client within a virtual environment filled with billions of entities was interested in a set of octree regions containing 10,000 entities, and was only really interested in 100 of those entities, it would take 0.62 seconds to update the client. Although this would not be an acceptable rate, there were several optimizations that were left out of the prototype to make this experiment a fair comparison.

One major optimization is to only dead reckon entities that are of possible interest. Currently all entities found within octree regions of interest are dead reckoned, and by using this optimization, a factor of three increase in overhead is realized for this particular simulation. Also, the packets were all of a uniform size of 1024 bytes. This is much bigger than needed; for example, DIS packets are 144 bytes, which would have saved a factor of 7 in memory copies. Also, although the computers used for this experiment were relatively high-end workstations, they do not perform as well on integer operations as a current day PC. Much of the update process within the client is integer based, which is directly related to CPU clock speed. It would not be surprising to purchase a home PC that ran at 600 MHz, and when compared to the 250 MHz workstations used within this experiment realize a factor of 2 increase in speed. Also, the code was compiled in a 'debug' state, so the compiler did not output optimized machine code.

Putting all of the optimizations together, it is not unrealistic to expect an almost 20 times increase in performance, bringing our example to 0.031 seconds per update, or about 30 Hz. This result, combined with the earlier result of a 1 Mbps bandwidth requirement, means that someone with a modern day computer and modern day Internet connection can participate in a virtual world with billions of entities, as long as he or she were only interested in about 100 entities at a time. Overall CPU and Bandwidth requirements for 100 interesting entities are plotted on a log scale as a function of entities within octree regions of interest in Figure 21.

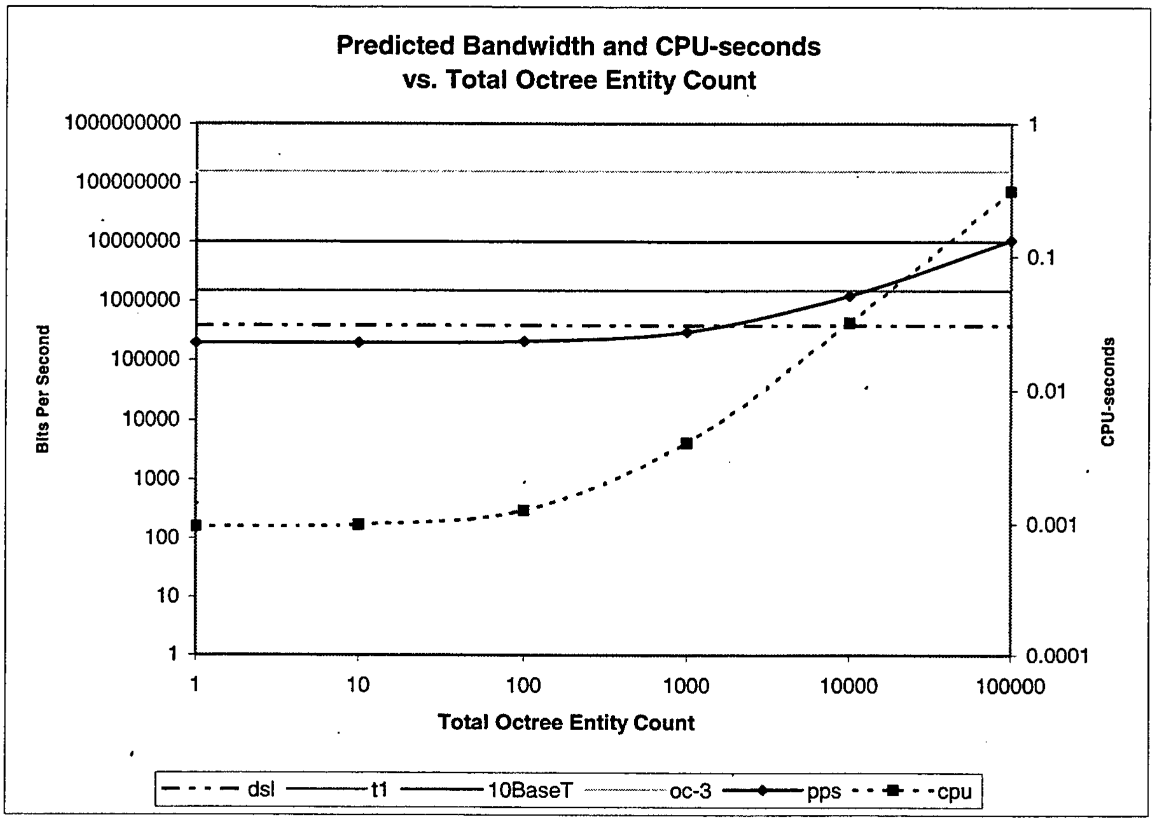


Figure 21. Predicted Bandwidth and CPU requirements versus total octree region entity count.

For example, using the computer and network technology available in a home today, a person can join a virtual environment containing a football stadium. He or she could sit in the stands among 10,000 other people, all of which would be updated via low fidelity messages sent to the octree regions. Of those 10,000 people, the nearest 100 of them would be updated at high fidelity messages sent via individual multicast address.

3. Octree Formulas

Figure 22 presents the average number of packets per second transmitted to each octree region using the Three-Tier system. These transmissions represent overhead for the Three-Tier system, and therefore it is best to reduce them. Chapter II presented an equation to predict worst-case packet overhead that each octree region would generate. Using this equation the goal was to keep packets per second per octree region below unity.

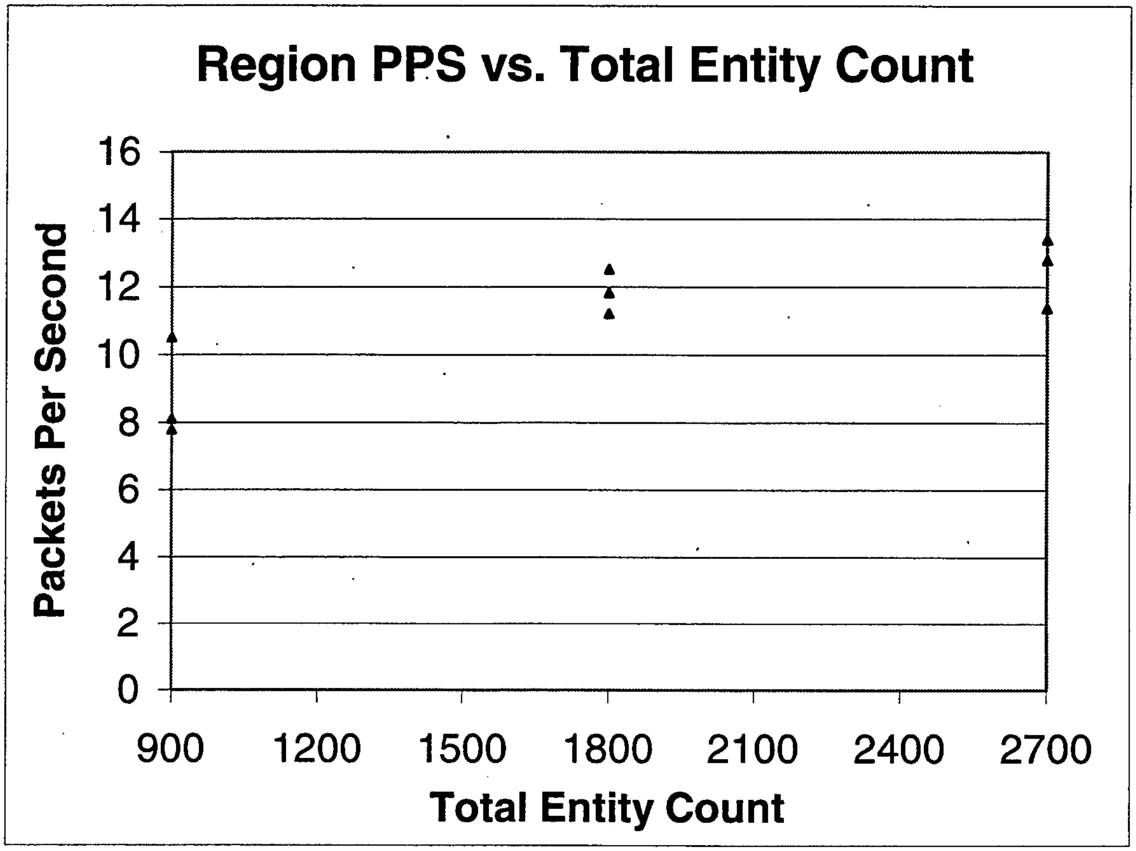


Figure 22. Raw Octree Region Packets Per Second vs. Total Entity Count

As can be seen above, the results were much higher than expected, but recall from section C that this scenario represents a worst case. Because each entity's smallest region was too large to allow subdivision of the octree, there were more entities than wanted in each region. To obtain the predicted results, the octree should have subdivided at 14 entities. By totaling the average packets per second from each region, dividing by the total number of entities within the environment, then multiplying by 14, we obtain a predicted result represented in Figure 23 below.

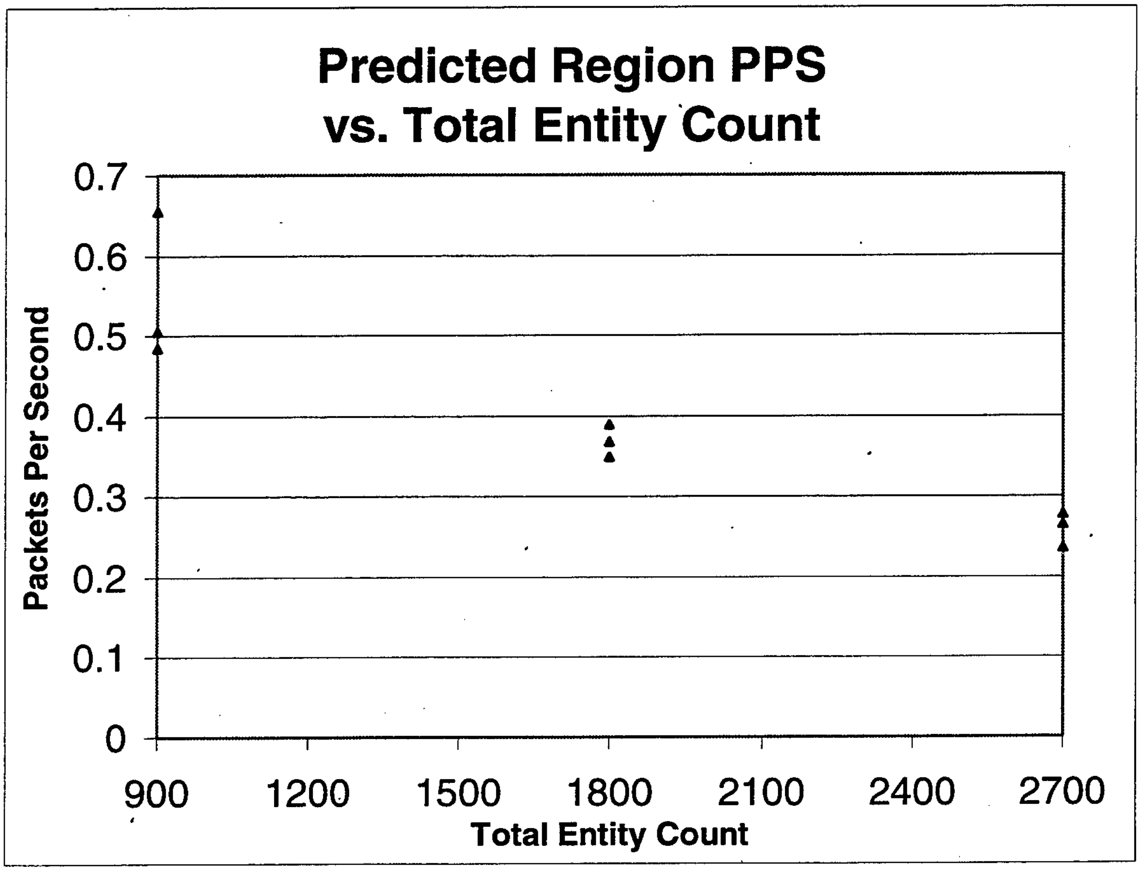


Figure 23. Predicted Octree Packets Second vs. Total Entity Count

These predicted results all fall below the predicted maximum of 1, the highest being 0.655. These predicted results confirm that the number 14 would have served well as an octree region entity cap.

Figure 24 below shows the minimum time an entity spent per region versus the total number of entities within the environment for the Three-Tier System.

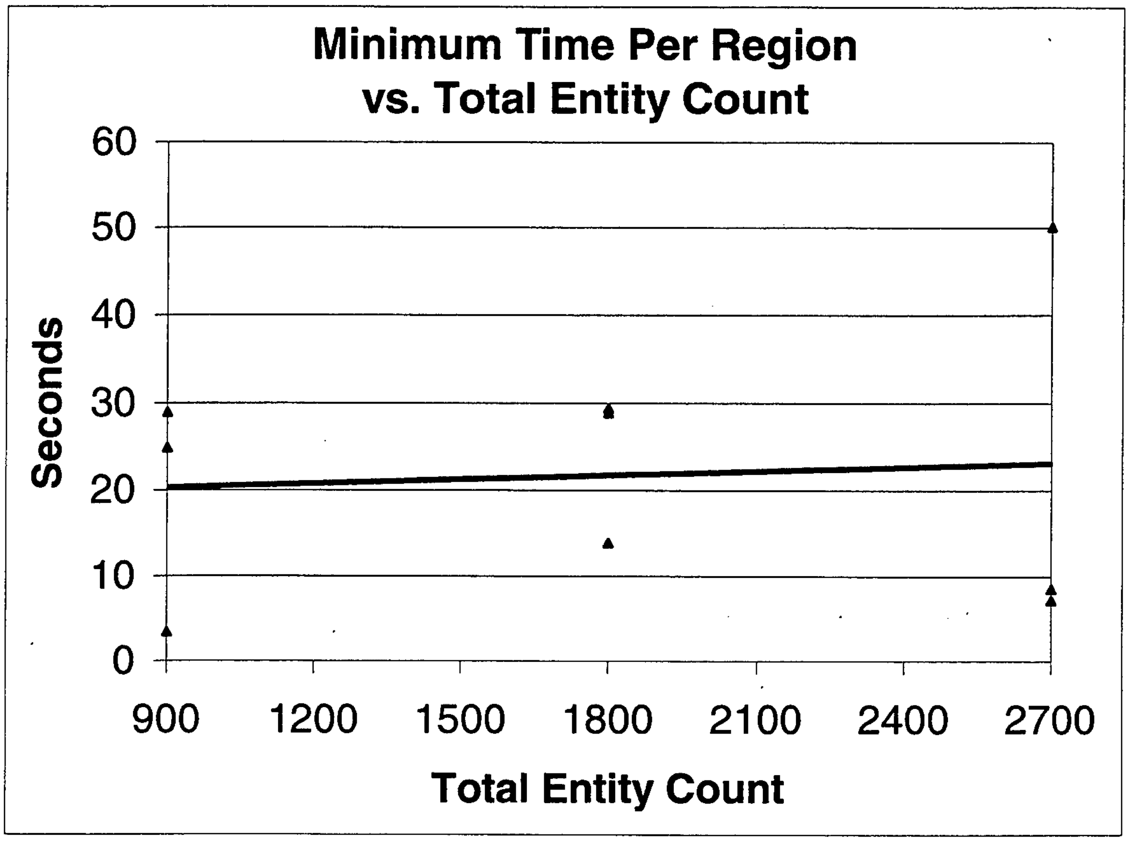


Figure 24. Raw Minimum Time Per Region vs. Total Entity Count

Using the formula to calculate minimum region size with the constant of ***T=300***, should have guaranteed within a probability of 50% that each entity stay within a region for at least 300 seconds. Again because of the extreme nature of the experiment, the raw results must be extrapolated to predict how the formulas would have worked in an average case. In this case, although the octree regions were slightly larger than each entity's smallest allowed region, the entities were confined to a 1000x1000x50 meter box, divided equally among four regions. By multiplying the results by the smallest region size of 7141, and dividing by the actual region size of 500 we derive the predicted results presented in Figure 25 below.

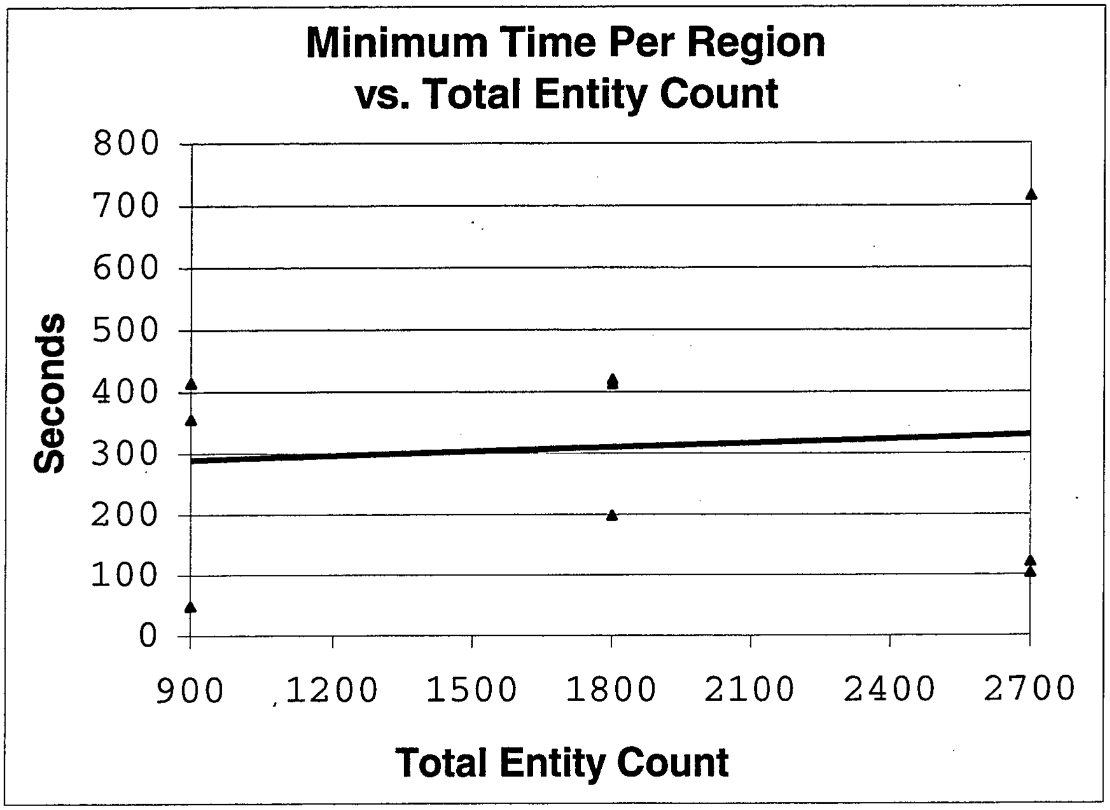


Figure 25. Predicted Minimum Time Per Region vs. Total Entity Count

Using these predicted results it can be seen that approximately 50% of the time the entity stayed within the region for more than 300 seconds, as the formula predicted.

These two results confirm that hypothesis H_5 is true, that is: The formulas presented in Chapter III work as expected, specifically $N_{\max} = P_{\max} cKT$ accurately predicts the maximum number of packets per second for a given region based on the number of entities, and $M = K(R + ST)$ predicts the average amount of time a given entity remains within one region based on region size.

4. Application Metric

The application used for this experiment was schooling fish. In theory the average number of fish in a school should not vary greatly between systems within a given total entity count as long as the CPU time allows for a reasonable amount of calculation. Because of the design of the fish entities, this school size is directly related to the number of interesting entities. Figure 26 below presents a chart of interesting entities versus the total number of entities within the virtual environment.

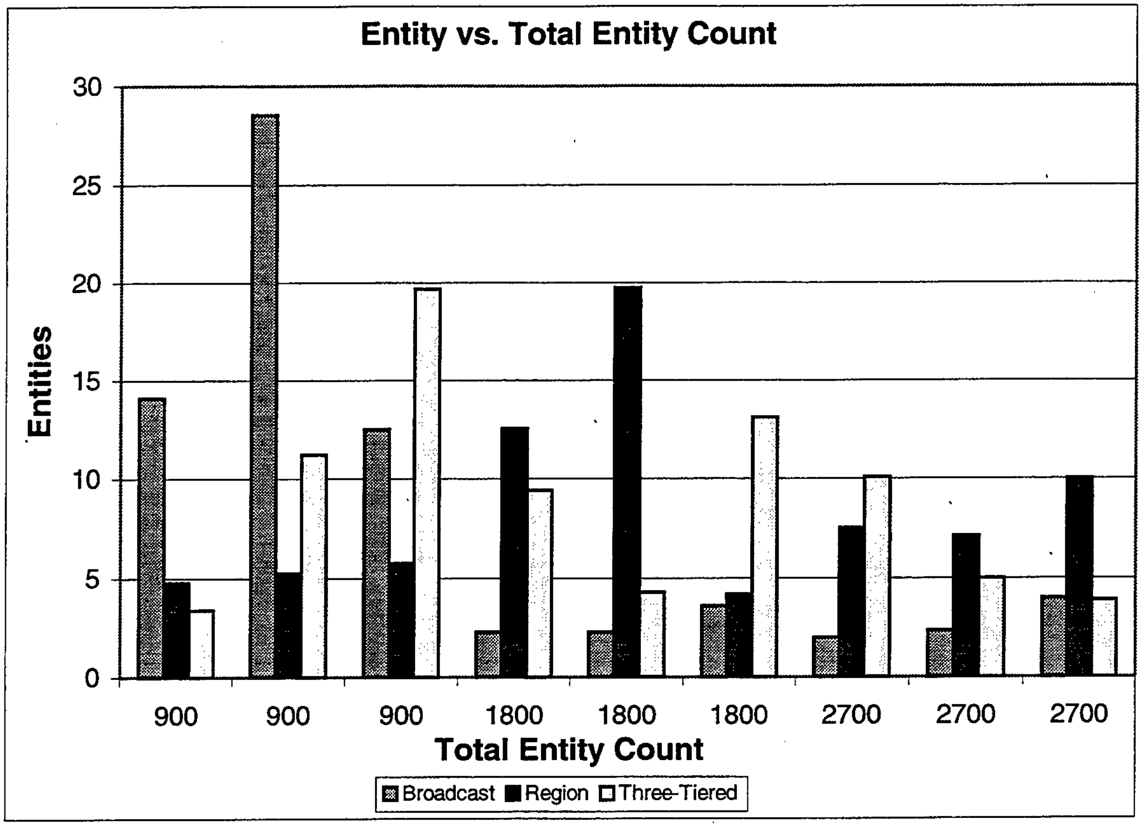


Figure 26. Interesting Entities vs. Total Entity Count

It can be seen from this chart that as the total entity count increases, and passes the 900 mark, the broadcast based system consistently has a lower number of interesting entities. It is no coincidence that after the 900 mark the broadcast based system shows this behavior, because this is the point where it does not have enough CPU cycles to update its position to other entities. The region and Three-Tier based systems however are not consistently different, as expected. These observations are backed by the statistics of F(2,6.148) with P=0.7002 for the comparison of the region and Three-Tier systems, showing statistically that they are not significantly different. For comparisons with the broadcast based systems however, P=0.0031 and P=0.0095, because of its failure.

These results confirm that hypothesis H_6 is true, that is: An application within a virtual environment using the Three-Tier Interest Management System works as well as the same application within the broadcast or region based systems. And in this case the Three-Tier system works better than the broadcast based system because of the latter's failure.

5. Networking

In theory the overall network behavior in terms of latency and packets per second should not vary between system architectures. Figure 27 shows the amount of network latency, in terms of seconds, versus the total number of entities within the virtual environment.

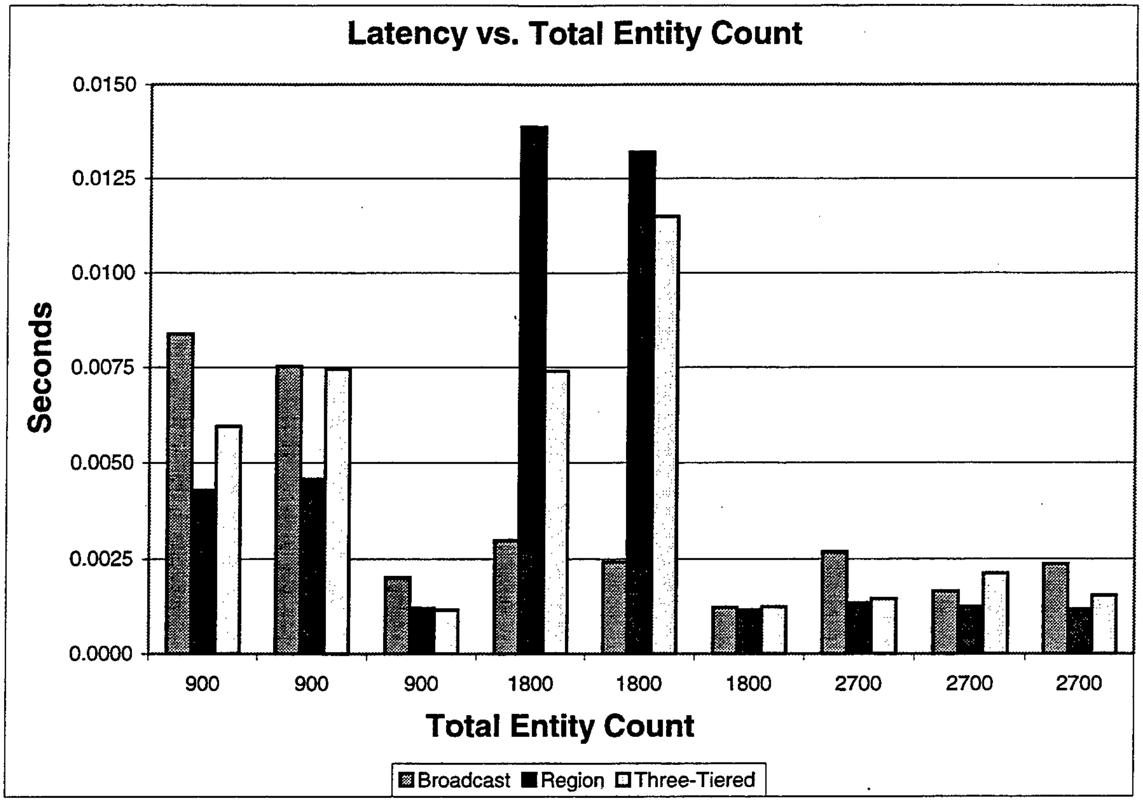


Figure 27. Network Latency vs. Total Entity Count

As expected, it is not clear from the chart that one system has a consistently lower or higher latency than any others. Statistically however, with F(2,8.901), P=0.0012 and P=0.002 for the comparisons between the broadcast and other systems, showing that the difference is significant. Although statistically significant, the difference in mean values is 0.98 and 1.12 milliseconds lower than the other systems, which is not an uncommon variation, and not detrimental to the simulation. This difference could be due to the fact that as entity count increased, the broadcast system did not produce as many packets per second as the other systems. For the difference between the Three-Tier and region based systems P=0.6532, which is not a significant difference.

Figure 28 shows the average number of packets per second transmitted on the local area network. By examining this graph, two facts can be inferred. First, all three systems showed reduced packet transmission as entity count increased, although the Three-Tier system showed this only mildly. The region based filtering system showed this a bit more, while the broadcast system showed this almost immediately.

Second, even if all three of the systems behaved perfectly, there was still a small overhead for the Three-Tier system. To show this more clearly, Figure 29 presents an enlargement of the lower left corner of Figure 28.

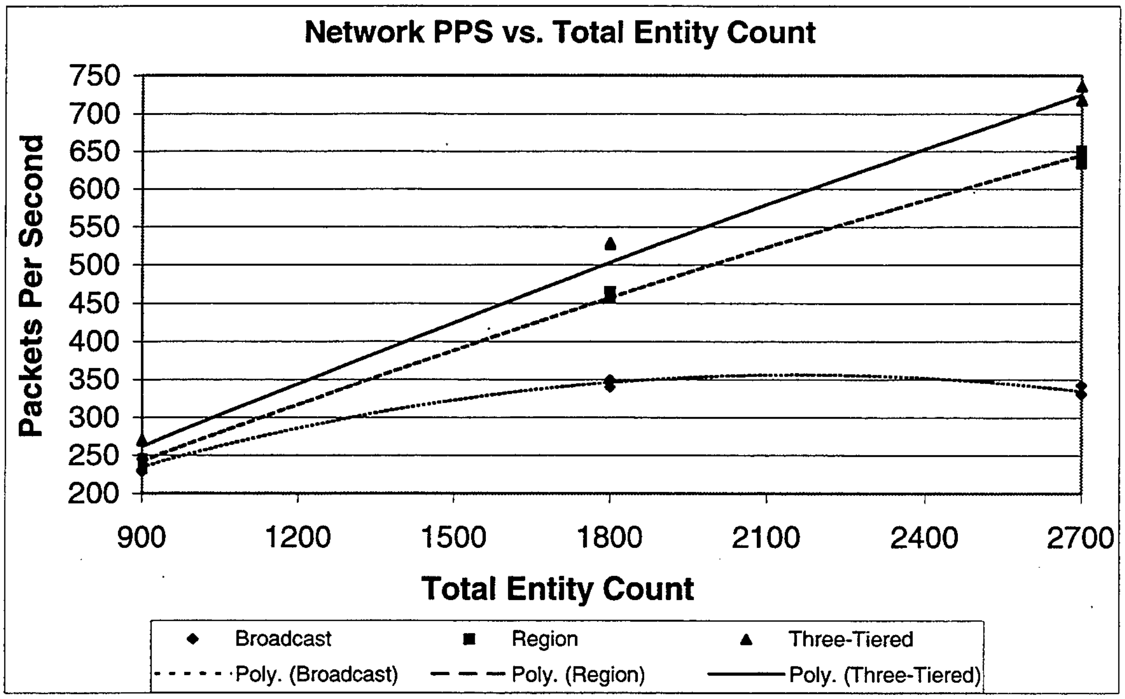


Figure 28. Total Network Packets Per Second vs. Total Entity Count

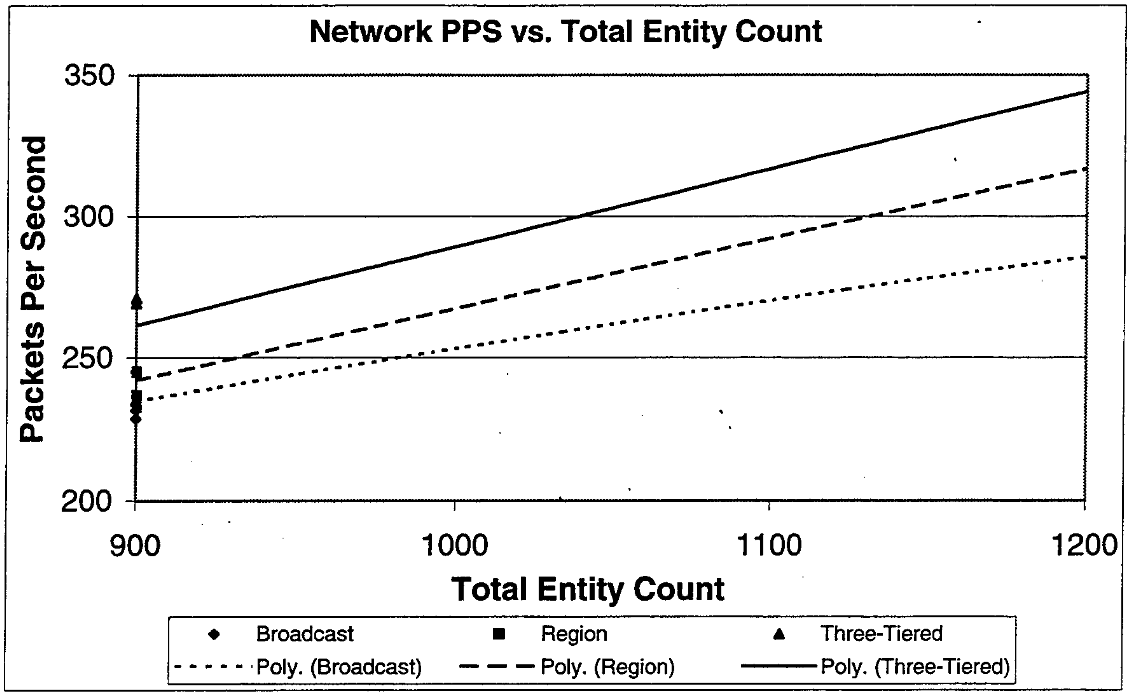


Figure 29. Enlargement of Network Packets Per Second vs. Total Entity Count

If one were to examine only the packet per second values at 900 total entities, it can be seen that the Three-Tier system transmitted approximately 20 more packets per second than the region-based system. This works out to be a total packet overhead of 0.022 packets per second per entity within the environment. This overhead is the result of two different causes. First, some of the packet overhead is most likely the result of multicast subscribe (IGMP) messages send out from each computer as it became interested in different entities. Second, each entity also transmits low fidelity messages to the octree regions, which were described earlier in subsection 3. While this overhead is measurable, each entity transmitting an extra packet once every 45 seconds is not significant to overall network performance.

If we combine the above with the fact that each packet was approximately the same size, then the Three-Tier system did not have a significant effect on overall network performance in terms of bandwidth either.

This result, combined with the earlier result from this subsection, has shown that hypothesis H_7 is true, that is: A virtual environment using the Three-Tier Interest Management System has approximately the same impact on the network in terms of packets per second, bandwidth, and latency, as broadcast or regions based systems.

E. SUMMARY

This chapter presented the experimental design and results of measurements comparing the Three-Tier interest management system against broadcast-based and region-based filtering. Specifically the results have shown that bandwidth, packets per second, and CPU time only depend upon the number of entities that a client is interested in, with a small overhead for entities within the regions of the octree that the client is interested in. The results have shown that the formulae from Chapter III accurately predict the packets per second per region of the octree as well as the average amount of time that each entity spends within a region. These results have also shown that architecture does not significantly affect network performance.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. IMPLICATIONS OF WORK

With the caveat that experimentation has not been conducted for all combinations of scenarios and formula constants, the work presented in this dissertation yields the following significant contributions to the field of interest management within networked virtual environments:

1. A system which scales in terms of CPU usage, packets per second, and bandwidth dependent on the number of entities a client is interested in, and is less dependent on the total number of entities within a virtual environment than broadcast or receiver-based systems. This, combined with the dynamic extendibility of the three-tired system, could allow the virtual environment to be persistent.

2. A formula that accurately predicts the amount of region switching overhead that a given entity will encounter. Because this formula is general it is thought that it can be used not only with the Three-Tier system but also to determine a pre-computed static size for region-based filtering.

3. A set of formulae that can be used to accurately determine when to split or merge a dynamically-sized region. Because these formulae are general it is

thought that they can be used not only with the Three-Tier system but also within dynamically sized region-based filtering systems.

B. LIMITATIONS OF WORK

More testing is needed on a much larger scale in terms of both time and entity count. To conduct even the simple experiment outlined in Chapter V with 10,000 entities requires as many as 50 computers connected together with a network properly configured for multicast, and with routers able to handle over 10,000 multicast routes. One real test of the Three-Tiered system's scalablity would be to properly implement it on a variety of operating systems, set up a dedicated computer to act as a server, and invite the Internet community in general to partake in the virtual environment. Some of this work is already underway, although certainly its adoption would be hindered by the lack of IP multicast support by many Internet Service Providers (ISP).

Much of the scalability presented in this dissertation still relies on IP multicast technology that is currently limited in many implementations. Chapter II outlined these concerns as well as current research efforts addressing these concerns. The work presented in this dissertation may be limited by any failures these research efforts may encounter. However, experimental results imply that some researchers working in the field of interest management may be solving problems that are too specific. Instead, perhaps they should focus their efforts on solving the general data distribution problems found within the network itself. For example, instead of solving the problem of grouping entities with like interests to save multicast addresses, each entity could be given one or more multicast addresses, and the more general problem of aggregating multicast routes within network routers could be solved instead.

The system presented in this dissertation is almost completely distributed. Along with distribution comes a lack of control. Implementers of virtual environments using this system may find difficulty in enforcing constraints or limiting participation in the environment. While this may not be necessary for large social environments, it is most definitely required for online gaming and other "pay-per-view" virtual environments. One possible solution for limiting participation may be to use the octree server as an encryption key server as well, and then encrypting all traffic within the octree regions.

In the current design and implementation of the Three-Tiered system a virtual environment must make assumptions about the type and number of dimensions that should make up the octree. To make the system more general a mechanism needs to be created to allow the tree to dynamically change the type and number of dimensions it represents. For example, many entities may need to filter based on positions and temperature. An 18-tree that represents these four dimensions could then be used as a first pass at the data instead of the octree, which represents only three of them.

Using the presented architecture, the virtual world must be of a fixed size, and cannot change in size dynamically. To fix this, the octree root node would need to be re-parented. Even so, the current dimensions of a region are transmitted in 64-bit floating-point format. To effectively solve the problem of a world of unlimited size, the system would need multiple frames of reference similar to those used in Spline (Barrus, Waters et al. 1996). One possible solution might be to create a "world of worlds" containing many virtual environments, each specifying their coordinate system relative to another world. Although this subject is perhaps outside the direct scope of interest management, without being able to grow the volume that a virtual environment encompasses, it cannot truly be called persistent or long-lived.

Currently a single server is used for octree consistency and searching. Although the server is lightweight and handles only a small number of requests, a fault-tolerant mechanism to distribute the workload among many servers must be created for truly scalable persistent virtual environments.

C. RECOMMENDATIONS FOR FUTURE WORK

In addition to addressing the limitations of this work, future work should address the fact that this is only one piece of a larger framework that must be in place in order to support a large, persistent virtual environment. A larger framework being worked on is called NPSNET-V, and is to be made up of the NPSNET-V Bamboo Foundation Modules: "Dynamic Protocols," "Interest Management" and "Persistent Universe." The goal is that together these Bamboo modules form the foundation for a persistent virtual environment inhabited by millions of entities, hosted around the world. This dissertation marks the completion of prototype versions of the "Dynamic Protocols" and "Interest Management" portions of NPSNET-V.

Persistent Universe adds the ability to store an entity's persistent state online and to find specific entities within the virtual environment. When an entity leaves a virtual environment, there may be a need to store its state just as it disconnects from the environment, so that it may reconnect and continue at some later time. One technology we are looking to exploit is the common Internet web server. Also, some past systems provided a method of finding entities within an environment, such as the Beacons in Spline (Barrus, Waters et al. 1996). To find entities, we are looking to dynamic DNS. This existing Internet infrastructure can be used by dynamically mapping host domain names to multicast addresses of entities, or address of regions within the environment.

Although this work attempts to address the problem of filtering data that is not needed by a client, it does not address how to decide which data is needed. Much work must go into the problem of automatically changing interest expressions in order to support very large, diverse, and dynamic virtual environments.

A similar, but more tractable problem is that, the current client subscribes to all regions that its spherical AOI intersects with, including all depths of the octree. Although this is not technically incorrect, there is more work to be done in the area of deciding what size regions are important. This type of modification to the current client combined with aggregation of entities can reduce bandwidth and CPU usage by a significant amount in large crowded virtual environments, providing a network equivalent to the functionality of the level-of-detail mechanism found in many scene graphs. Although it is imagined that efficient aggregation can be used in the larger regions, this also must be demonstrated in future work.

Experimentation presented in this dissertation seems to indicate that the Three-Tier system will work well for large virtual environments, but research needs to be conducted to see if this architecture will work well for small collaborative virtual environments, for example a medical surgery simulation.

D. SUMMARY

This dissertation has shown that it is possible to create an interest management software architecture that will allow persistent distributed virtual environments to scale dependent only on the number of entities a client is interested in at any one time.

The formulae presented in this work can accurately predict the maximum number of packets per second for a given region, as well as the average amount of time that a given entity remains within one region. These formulas can be used with not only the Three-Tier system, but also existing region-based interest management systems.

These findings must be prefaced with the fact that experimentation has not been conducted for all combinations of scenarios and formula constants. Nevertheless, these are significant contributions to the field of interest management and are expected to help make the dream of a single persistent virtual environment inhabited by millions of entities come true.

LIST OF REFERENCES

Abrams, H., K. Watsen, et al. (1998). Three-Tiered Interest Management for Large-Scale Virtual Environments. ACM Symposium on Virtual Reality Software and Technology, Taipei, Taiwan. pp. 125-129.

Barrus, J. W., R. C. Waters, et al. (1996). TR-95-16a, Locales and Beacons: Efficient and Precise Support For Large Multi-User Virtual Environments. Cambridge, Mitsubishi Electric Information Technology Center America. http://www.merl.com/reports/TR95-16a/locales.html

Broll, W. (1997). Populating the Internet: Supporting Multiple Users and Shared Applications with VRML. VRML 97, Monterey, CA. pp. 33-40.

Calvin, J. O., D. P. Cebula, et al. (1995). Data Subscription in Support of Multicast Group Allocation. 13th Workshop on Standards for the Interoperability of Distributed Simulations. pp. 593-594.

Carlsson, C. and O. Hagsand (1993). DIVE - A Multi-User Virtual Reality System. IEEE Virtual Reality Annual International Symposium. pp. 394-400.

Carlsson, C. and O. Hagsand (1993). "DIVE - A Platform For Multi-User Virtual Environments." Computers & Graphics **17**(6): 663-669.

Casner, S., D. M. Kristol, et al. (1997). Frequently Asked Questions (FAQ) on the Multicast Backbone (MBONE). http://www.sc.columbia.edu/~hgs/internet/mbone-faq.html.

Dahmann, J., R. Weatherly, et al. (1999), Creating Computer Simulation Systems: An Introduction to the High Level Architecture. Upper Saddle River, NJ: Prentice-Hall 1999.

Das, T. K., G. Singh, et al. (1997). Developing Social Virtual Worlds using NetEffect. Sixth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, MIT, Cambrige Massachusetts. pp. 148-154.

Durand, A. and B. Buclin (1999). RFC2546, 6Bone Routing Practice. http://ieft.org/rfc/rfc2546.txt

Fink, B. (1999). What is the 6bone. April 11, 1999 http://www.6bone.net/about_6bone.html

Floyd, S., V. Jacobson, et al. (1997). "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." ACM Transactions on Networking 5(6): 784-803.

Funkhouser, T. A. (1995). RING: A Client-Server System for Multi-user Virtual Environments. Symposium on Interactive 3-D Graphics, Monterey, CA USA. pp. 85-92.

Greenhalgh, C. (1999). MASSIVE-3 / HIVEK Introduction. Slide presentation.

Greenhalgh, C. and S. Benford (1995). MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading. IEEE 15th International Conference on Distributed Computing Systems, Vancouver, Canada, IEEE Computer Society. pp. 27-34.

Greenhalgh, C. and S. Benford (1997). Boundaries, Awareness and Interaction in Collaborative Virtual Environments. Sixth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, MIT, Cambrige Massachusetts. pp. 193-198.

Handley, M., D. Thaler, et al. (1999). The Internet Multicast Address Allocation Architecture. July 2,1999. http://search.ietf.org/internet-drafts/draft-ietf-malloc-arch-03.txt

Harless, G. J. and R. V. Rogers (1995). Achieving O(N) in Simulating the Billiards Problem in Discrete-Event Simulation. 1995 Winter Simulation Conference, Orlando, FL. pp. 751-756.

Hubbold, R., X. Dongbo, et al. (1996). MAVERIK - the Manchester Virtual Environment Interface Kernel. 3rd Eurographics Workshop on Virtual Environments, Monte-Carlo.

Kahney, L. (1999). Net's Change of Address. Wired News. http://www.wired.com/news/news/technology/story/20765.html. July 15, 1999.

Kazman, R. (1995). "HIDRA: An Architecture for Highly Dynamic Physically Based Multi-Agent Simulations." International Journal in Computer Simulation 5, 1995. pp. 149-166

Kazman, R. (1993). Making WAVES: On the Design of Architecture for Low-end Distributed Virtual Environments. IEEE Virtual Reality Annual International Symposium, pp. 443-449.

Kazman, R. (1995). Load Balancing, Latency Management and Separation Concerns in a Distributed Virtual World. Parallel Computations -- Paradigms and Applications. A. Zomaya, Chapman & Hall.

Lea, R., Y. Honda, et al. (1997). Community Place: Architecture and Performance. VRML 97, Monterey, CA. pp. 41-50.

Levine, B. N., J. Crowcroft, et al. (1999). Consideration of Receiver Interest for IP Multicast Delivery. Unpublished Paper.

Lim, M. and D. Lee (1999). Scalable Region Management for Distributed Virtual Environments. Unpublished Paper.

Macedonia, M. (1995). A Network Software Architecture For Large Scale Virtual Environments. Ph.D. Dissertation, Naval Postgraduate School. Monterey, CA.

Macedonia, M. R. and D. P. Brutzman (1994). "MBone Provides Audio and Video Across the Internet." IEEE Computer **27**(4): 30-36.

Macedonia, M. R., M. J. Zyda, et al. (1994). "NPSNET: A Network Software Architecture for Large Scale Virtual Environments." Presence **3**(4): 265-287.

Macedonia, M. R., M. J. Zyda, et al. (1995). "Exploiting Reality with Multicast Group: A Network Architecture for Large-scale Virtual Environments." IEEE Computer Graphics & Applications(September 1995): 38-45.

Makbily, Y., C. Gotsman, et al. (1999). Geometric Algorithms for Message Filtering in Decentralized Virtual Environments. Symposium on Interactive 3D Graphics, Atlanta Georgia. pp. 39-46.

Mastagllio, T. W. and R. Callahan (1995). "A large-Scale Complex Virtual Environment for Team Training." IEEE Computer **28**(7): 49-56.

McGregor, D. (1998). CPU Performance and Multicast. Monterey, Naval Postgraduate School. http://www.stl.nps.navy.mil/~mcgredo/projectNotebook/mcast/ethernet.html.

Mellon, L. and D. West (1995). Architectural Optimizations to Advanced Distributed Simulation. 1995 Winter Simulation Conference. pp. 634-641.

Morse, K. L. (1998). Dynamic Interest Management Using Mobile Agents, Unpublished Paper.

Morse, K. L. (1999). An Adaptive, Distributed Algorithm for Interest Management,.Slide presentation.

Morse, K. L. (1999). An Adaptive, Distributed Algorithm for Interest Management, Unpublished Dissertation Proposal.

Morse, K. L., L. Bic, et al. (2000). "Interest Management in Large-Scale Virtual Environments." To be published in Presence **9**(1).

Multicast-Implementation-Study (1999). Reliable Multicast Protocols. http://www.tascnets.com/mist/doc/mcpCompare.html.

Network Instruments (1998). Observer Users Manual. Minneapolis.

Oran, D. (1999). Minutes of the July 1999 IETF MAESTRO BoF. http://ietf.org/proceedings/99jul/45th-99jul-ietf-101.html

Pettifer, S. R. (1997). Deva: a Coherent Operating Environment for Large Scale Virtual Reality Applications. Presented at Virtual Reality Universe 1997. http://www.cyberedge.com/vru_papers/pettifer.htm

Pettifer, S. R. (1999). An Operating Environment for Large Scale Virtual Reality. Ph.D. Dissertation, University of Manchester.

Powell, E. T., L. Mellon, et al. (1996). Joint Precision Strike Demonstration (JPSD) Simulation Architecture. 14th Workshop on Standards for the Interoperability of Distributed Simulations. pp. 807-810.

Pulkka, A. K. (1995). Spatial Culling of Interpersonal Communications within Large-Scale Multi-User Environments. Master's Thesis, University of Washington.

Rak, S. J. and D. J. Van Hook (1996). Evaluation of Grid-Based Relevance Filtering for Multicast Group Assignment. 14th Workshop on Standards for the Interoperability of Distributed Simulations. pp. 739-747.

Reynolds, C. W. (1987). Flocks, Herds, and Schools: A Distributed Behavioral Model. SIGGRAPH '87 Conference Proceedings, Anaheim, California, July 27-31. pp. 25-34.

Shaw, C., M. Green, et al. (1993). "Decoupled Simulation in Virtual Reality with the MR Toolkit." ACM Transactions on Information Systems **11**(3): 287-317.

Singh, G., L. Serra, et al. (1994). "BrickNet: A Software Toolkit for Networks-Based Virtual Worlds." Presence: Teleoperators and Virtual Environments **3**(1): 19-34.

Singhal, S. K. and D. R. Cherition (1996). Using Projection Aggregations to Support Scalability in Distributed Simulation. 16th International Conference on Distributed Computing Systems, Hong Kong. pp. 196-206.

Smith, J. E., K. L. Russo, et al. (1995). Prototype Multicast IP Implementation in ModSAF. 12th Workshop on Standards for the Interoperability of Defense Simulations, Orlando, Florida. pp. 175-178.

Snowdon, D. N. and A. J. West (1994). "AVIARY: Design Issues for Future Large-Scale Virtual Environments." Presence 3(4): 288-308.

Stardust.com (1999). QoS Protocols & Architectures. Campbell, California, Stardust.com. http://www.qosforum.com/white-papers/qosprot_v3.pdf.

Steinman, J. S. and F. Wieland (1994). Parallel Proximity Detection and the Distributed List Algorithm. The 1994 Workshop on Parallel and Distributed Simulation. pp. 3-11.

Stevens, W. R. (1998). Networking APIs: Sockets and XTI. New Jersey, Prentice Hall.

Sugano, H., K. Otani, et al. (1997). SpaceFusion: A Multi-Server Architecture For Shared Virtual Environments. VRML 97, Monterey, CA. pp. 51-58.

Teitelbaum, B. (1999). Draft QBone Architecture. http://www.internet2.edu/qos/wg/papes/qbArch/1.0/draft-i2-qbone-arch-1.0.html

Thaler, D. and B. Aboba (1998). Multicast Debugging Handbook. http://search.ietf.org/internet-drafts/draft-ietf-mboned-mdh-01.txt.

Thaler, D., D. Estrin, et al. (1998). Border Gateway Multicast Protocol (BGMP): Protocol Specification. http://search.ietf.org/internet-drafts/draft-ietf-idmr-gum-04.txt

Van Hook, D. J., J. O. Calvin, et al. (1994). An Approach to DIS Scaleability. 11th Workshop on Standards for the Interoperability of Distributed Simulations. pp. 347-456.

Van Hook, D. J., S. J. Rak, et al. (1994). Approaches to Relevance Filtering. 11th Workshop on Standards for the Interoperability of Distributed Simulations. pp. 367-369.

Van Hook, D. J., S. J. Rak, et al. (1996). Approaches to RTI Implementation of HLA Data Distribution Management Services. 96-14-084, 15th Workshop on Standards for the Interoperability of Distributed Simulations.

Vellon, M., K. Marple, et al. (1998). The Architecture of a Distributed Virtual Worlds System. Redmond, Microsoft Research. http://www.research.microsoft.com/vwg/

Wang, Q., M. Green, et al. (1995). EM - An Environment Manager For Building Networked Virtual Environments. IEEE Virtual Reality Annual International Symposium. pp. 11-19.

Waters, R. C., D. B. Anderson, et al. (1997). Design of the Interactive Sharing Transfer Protocol. Sixth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, MIT, Cambrige Massachusetts. pp. 140-147.

Watsen, K. and M. Zyda (1998). Bamboo - A Portable System for Dynamically Extensible, Real-Time, Networked, Virtual Environments. IEEE Virtual Reality Annual International Symposium, Atlanta, Georgia. pp. 252-259.

Watsen, K. and M. Zyda (1998). Bamboo - Supporting Dynamic Protocols for Virtual Environments. IMAGE 98, Scottsdale, Arizona. KA-1-9

Wilson, A. and R. Weatherly (1994). New Traffic Reduction and Management Tools for ALSP Confederations. Elecsim Internet Conference.

APPENDIX A. THREE-TIERED SYSTEM SOURCE CODE

A. INTRODUCTION

This appendix contains source code for the Three-Tiered interest management system used in experiment described in Chapter V. Presented below is the source-code for the Bamboo module "IMServerModule" containing the lightweight server and the module "IMBaseModule" that contains the four main base classes.

This source-code presented here is only for completeness, will not work on many platforms, however, up to date platform-independent versions of these modules are available from the NPSNET-V website (http://npsnet.org/~npsnet/v).

B. IMSERVERMODULE

1. server.h

```
#define CLIENT_BASE_ADDRESS 0
#define SERVER_BASE_ADDRESS 0

#define LEAF 0
#define BRANCH 1

typedef struct Octree
{
 char type;
 char lock;
 Octree *parent;
 Octree *children;
 unsigned int address;
} my_o;
```

2. server.c++

```
#include "server.h"
#include "IMPacket.h"
#include "IMCellRegion.h"

#include <values.h>
#include <stdlib.h>
```

```
#include <stdio.h>

#include <sys/time.h>

#include <unistd.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

#include <net/if.h>
#include <net/soioctl.h>

#include "set.h"
#include "list.h"

#include "bbThread.h"

#define SERVER_PORT 9805
#define MAX_REGIONS 1000000
#define WORLD_SIZE 10000.0
#define SPLIT_LIMIT 30
#define MERGE_LIMIT 10
#define DELAY_TIME 20.0

Octree myalloc[MAX_REGIONS];
Octree *hash[MAX_REGIONS];
Octree *tree;
unsigned long base_address;
int pt = 0;

int bigread(int fd,void *buf,int num)
{
 int cnt = 0;
 while(cnt < num && cnt != -1) cnt += read(fd,(char *)buf+cnt,num-cnt);
 return cnt;
}

int bigwrite(int fd,void *buf,int num)
{
 int cnt = 0;
 while(cnt < num && cnt != -1) cnt += write(fd,(char *)buf+cnt,num-cnt);
 return cnt;
}

unsigned int allocServerAddress()
{
 static unsigned long count = base_address;
 return count++;
}

int rc = 0;

// Cell alloctaion
Octree *newCells(Octree *parent,unsigned int newaddr[8])
{
 Octree *cells = &myalloc[rc];
 rc+=8;
 for(int i=0; i<8; i++)
     {
      cells[i].lock = 0;
      cells[i].address = newaddr[i];
      cells[i].num_people = 0;
      cells[i].type = LEAF;
      cells[i].parent = parent;
     }

 return cells;
}

// Given a LEAF, make it a BRANCH, and
// give it 8 LEAF children
void splitTree(Octree *cell,unsigned int newaddr[8])
{
 int n = cell->num_people;
 pt -= cell->num_people;

 cell->type = BRANCH;
 cell->children = newCells(cell,newaddr);

 struct in_addr grpaddr;
 grpaddr.s_addr = cell->address;
 printf("splitting cell %s\n",inet_ntoa(grpaddr));
}

// Given a Parent BRANCH, with only LEAF children
// make Parent a LEAF
void mergeTree(Octree *cell)
{
 printf("Not enough people, joining cells\n");

 cell->type = LEAF;
 cell->num_people = 0;
 cell->children = NULL;
}

Octree *AddresstoCell(Octree *mytree,unsigned long address)
{
 if (mytree->address == address) return mytree;
 if (mytree->type == LEAF) return NULL;

 Octree *ret = NULL;
 if (!ret) ret = AddresstoCell(&mytree->children[0],address);
 if (!ret) ret = AddresstoCell(&mytree->children[1],address);
 if (!ret) ret = AddresstoCell(&mytree->children[2],address);
 if (!ret) ret = AddresstoCell(&mytree->children[3],address);
 if (!ret) ret = AddresstoCell(&mytree->children[4],address);
 if (!ret) ret = AddresstoCell(&mytree->children[5],address);
 if (!ret) ret = AddresstoCell(&mytree->children[6],address);
 if (!ret) ret = AddresstoCell(&mytree->children[7],address);

 return ret;
}

int XYZRtoCells(Octree *mytree,Extent *bound,struct CellRegion *regs)
{
 int c = 0;

 double r2 = regs[0].ext.r/2.0;
double cx = regs[0].ext.x;
double cy = regs[0].ext.y;
double cz = regs[0].ext.z;

//Always return parent.
CellRegion *origregs = regs;
regs[0].address = mytree->address;
regs = &regs[1];
int count = 1;

unsigned char mask = 0xFF;
if (bound->x > cx+bound->r) mask&=240; //11110000
else if (bound->x < cx-bound->r) mask&=15; //00001111
if (bound->y > cy+bound->r) mask&=204; //11001100
else if (bound->y < cy-bound->r) mask&=51; //00110011
if (bound->z > cz+bound->r) mask&=170; //10101010
else if (bound->z < cz-bound->r) mask&=85; //01010101

if (0x1&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx - r2;
    regs[0].ext.y = cy - r2;
    regs[0].ext.z = cz - r2;

    if (mytree->children[0].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[0],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[0].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }

if (0x2&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx - r2;
    regs[0].ext.y = cy - r2;
    regs[0].ext.z = cz + r2;

    if (mytree->children[1].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[1],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[1].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }
```

```
if (0x4&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx - r2;
    regs[0].ext.y = cy + r2;
    regs[0].ext.z = cz - r2;

    if (mytree->children[2].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[2],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[2].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }

if (0x8&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx - r2;
    regs[0].ext.y = cy + r2;
    regs[0].ext.z = cz + r2;

    if (mytree->children[3].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[3],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[3].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }

if (0x10&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx + r2;
    regs[0].ext.y = cy - r2;
    regs[0].ext.z = cz - r2;

    if (mytree->children[4].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[4],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[4].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }

if (0x20&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx + r2;
    regs[0].ext.y = cy - r2;
    regs[0].ext.z = cz + r2;

    if (mytree->children[5].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[5],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[5].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }

if (0x40&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx + r2;
    regs[0].ext.y = cy + r2;
    regs[0].ext.z = cz - r2;

    if (mytree->children[6].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[6],bound,regs);
        regs = &regs[c];
        count += c;
       }
    else
       {
        regs[0].address = mytree->children[6].address;
        regs[0].below = 0;
        regs = &regs[1];
        count++;
       }
   }

if (0x80&mask)
   {
    regs[0].ext.r = r2;
    regs[0].ext.x = cx + r2;
    regs[0].ext.y = cy + r2;
    regs[0].ext.z = cz + r2;

    if (mytree->children[7].type == BRANCH)
       {
        c = XYZRtoCells(&mytree->children[7],bound,regs);
        regs = &regs[c];
```

```
         count += c;
        }
     else
        {
         regs[0].address =mytree->children[7].address;
         regs[0].below = 0;
         regs = &regs[1];
         count++;
        }
    }

 //Set number of regions below this one.
 origregs->below = count-1;
 //Return the number of regions.
 return count;
}

//Given XYZ return cell pointer and extent
Octree *XYZtoCell(double x,double y,double z)
{
 int i;
 double xa,ya,za,l,r,t,b,n,f;
 r=t=f= WORLD_SIZE;
 l=b=n=-WORLD_SIZE;
 Octree *cell = tree;

 while(cell->type == BRANCH)
      {
       xa = (l+r)/2.0;
       ya = (b+t)/2.0;
       za = (n+f)/2.0;

       if (x<=xa)
          {
           r=xa;
           if (y<=ya)
              {
               t=ya;
               if (z<=za)
                  {
                   i=0;
                   f=za;
                  }
               else
                  {
                   i=1;
                   n=za;
                  }
              }
           else
              {
               b=ya;
               if (z<=za)
                  {
                   i=2;
                   f=za;
                  }
               else
                  {
                   i=3;
                   n=za;
                  }
              }
          }
       else
          {
           l=xa;
           if (y<=ya)
              {
               t=ya;
               if (z<=za)
                  {
                   i=4;
                   f=za;
                  }
               else
                  {
                   i=5;
                   n=za;
                  }
              }
           else
              {
               b=ya;
               if (z<=za)
                  {
                   i=6;
                   f=za;
                  }
               else
                  {
                   i=7;
                   n=za;
                  }
              }
          }

       cell = &cell->children[i];
      }

 return cell;
}

static bbThread *loopThread;
static int s2;

void loopFunc(bbThread *,bbData *that)
{
 ServerPacket packet;

 //wait for new connection
 fd_set listenfds;
 FD_ZERO(&listenfds);
 FD_SET(s2, &listenfds);
 struct timeval wait = {1,0};
 select(s2+1,&listenfds,NULL,NULL,&wait);
 if (!FD_ISSET(s2,&listenfds)) return;

 //accept new TCP connection
 static struct sockaddr sa_client;
 static int cs=sizeof(struct sockaddr);
 int s = accept(s2,&sa_client,&cs);
```

```
if (read(s,&packet,sizeof(ServerPacket)) == -1) perror("read");

switch(packet.type)
      {
       case ServerPacket::MERGE:
            {
             int lock = 0;
             printf("Client merging cell %d!\n",packet.split.address);
             Octree *cell = AddresstoCell(tree,packet.split.address);
             if (cell && cell->type == BRANCH)
                {
                 int ok = 1;
                 for(int i=0;i<8;i++)
                    if (cell->children[i].type == BRANCH ) ok = 0;
                 if (ok)
                    {
                     mergeTree(cell);
                     lock = 1;
                    }
                 write(s,&lock,sizeof(lock));
                 printf("Unlocking cell %d!\n",packet.split.address);
                 cell->lock = 0;
                }
             else write(s,&lock,sizeof(lock));
            } break;

       case ServerPacket::SPLIT:
            {
             int lock = 0;
             printf("Splitting cell %d!\n",packet.split.address);
             Octree *cell = AddresstoCell(tree,packet.split.address);
             if (cell)
                {
                 if (cell->type == LEAF)
                    {
                     splitTree(cell,packet.split.newaddr);
                     lock = 1;
                    }
                 write(s,&lock,sizeof(lock));
                 printf("Unlocking cell %d!\n",packet.split.address);
                 cell->lock = 0;
                }
             else write(s,&lock,sizeof(lock));
            } break;

       case ServerPacket::LOCK:
            {
             int lock = 0;
             printf("Trying to lock cell %d!\n",packet.split.address);
             Octree *cell = AddresstoCell(tree,packet.split.address);
             if (cell)
                {
                 if (!cell->lock)
                    {
                     lock = 1;
                     printf("Locking cell %d!\n",packet.split.address);
                     cell->lock = 1;
                    }
                }
             else printf("Cell %d not found!\n",packet.split.address);
             write(s,&lock,sizeof(lock));
             } break;

        case ServerPacket::MALLOC:
             {
              static unsigned int addrs[256];
              for(int i=0;i<packet.malloc.num;i++)
                 addrs[i]=allocServerAddress();
              write(s,addrs,sizeof(unsigned int)*packet.malloc.num);
              printf("allocated %d addresses\n",packet.malloc.num);
             } break;

        case ServerPacket::SEARCH:
             {
              //do sphere-octree intersection.
              Extent ext;
              static struct CellRegion regs[MAX_REGIONS];

              regs[0].ext.r = WORLD_SIZE;
              regs[0].ext.x = regs[0].ext.y = regs[0].ext.z = 0;
              ext.x=packet.search.x; ext.y=packet.search.y;
              ext.z=packet.search.z; ext.r=packet.search.r;
              int n = XYZRtoCells(tree,&ext,regs);
              write(s,&n,sizeof(int));
              bigwrite(s,regs,sizeof(CellRegion)*n);
             } break;

        default: printf("ERROR! Unknown message type %d\n",packet.type);
       }

 close(s);
}

extern "C"
{
void initFunc()
{
 printf("Starting server with world size set to %f\n",WORLD_SIZE);
 base_address=inet_addr("239.0.0.1");

 for (int i=0;i<MAX_REGIONS;i++) hash[i]=0;

 tree = (Octree *)malloc(sizeof(Octree));

 // Start us off w/ 8.
 tree->type = LEAF;
 tree->address = allocServerAddress();
 tree->num_people = 0;
 tree->lock = 1; // We never want this to split.
 hash[tree->address-base_address] = tree;
 unsigned int addrs[8];
 for(i=0;i<8;i++) addrs[i]=allocServerAddress();
 splitTree(tree,addrs);

 //Set up our incoming port
 struct in_addr bind_address;
 bind_address.s_addr = htonl(INADDR_ANY);

 struct sockaddr_in sa;
 sa.sin_family=AF_INET;
 sa.sin_addr=bind_address;
 sa.sin_port=htons(SERVER_PORT);
 s2 = socket(PF_INET,SOCK_STREAM,0);
 if (bind(s2,(struct sockaddr *)&sa,sizeof(struct sockaddr_in)) == -1)
     perror("bind");
 listen(s2,1000);

 loopThread = new bbThread(loopFunc,NULL);
}

void exitFunc(void)
{
 printf("Shutting down server....\n");
 delete loopThread;
 printf("Done.\n");
}
} //extern "C"
```

C. IMBASECLIENTMODULE

1. IMbaseClient.h

```
#ifndef _IMBASECLIENT
#define _IMBASECLIENT
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <net/if.h>
#include <net/soioctl.h>
#include <arpa/inet.h>

#include <multimap.h>
#include <set.h>
#include "bbMutex.h"
#include "IMBaseEntity.h"
#include "IMBaseOutEntity.h"

//Forward declare classes instead of including them
//This saves LOTS of memory when using Bamboo modules
class bbThread;
class Msocket;
class IMBaseInEntity;
class IMBaseProtocol;
class IMNode;

#define DEFAULT_SERVER_NAME "sse.cs.nps.navy.mil"
#define DEFAULT_SERVER_PORT 9805
#define DEFAULT_CLIENT_PORT 9905
#define MAX_READ_REGIONS 4096

class IMBaseClient
{
 public:
  static  void readFunc(bbThread *, bbData *);
  virtual void mcastcheck();

  bbMutex tree_mutex;
public:

  int bigread(int fd,void *buf,int num)
  {
   int cnt = 0;
   while(cnt < num && cnt != -1)
         cnt += read(fd,(char *)buf+cnt,num-cnt);
   return cnt;
  }

  int bigwrite(int fd,void *buf,int num)
  {
   int cnt = 0;
   while(cnt < num && cnt != -1)
         cnt += write(fd,(char *)buf+cnt,num-cnt);
   return cnt;
  }

  int s2;           //mcast socket for sending
  int ourtcpsocket; //tcp socket waiting for connections

  IMNode *tree;

  IMBaseProtocol *base_proto;

  struct sockaddr_in sa;    // sockaddr_in used to connect to server
  char server_name[255];
  short server_port;

  void setup();

  bool auto_tick;      // Should we automaticly tick outentities?
  bool internal_clock; // Should we set our time based on gettimeofday?
  double simtime;      // Simulation Time

  // Set of all out-bounds
  set<IMBaseOutEntity *,IMBaseEntity::compare> outlist;

  // Set of all in-bounds
  set<IMBaseInEntity *,IMBaseEntity::compare> inlist;

  bbThread *readThread;

  virtual void filter(McastPacket &mpacket,IMBaseInEntity *ng);

  typedef pair<unsigned int,McastPacket> packet_type;
  multimap<double,packet_type> delaySend;
  multimap<double,unsigned int> delaySplit,delayMerge;
  multimap<double,IMBaseOutEntity *> delayRespond;

  unsigned int clientaddress;

public:

  float k1; //constant for smallest region formula
  float k2; //constant for dead reckoning formula
  float MinTimeInRegion; //Minimum time to spend in a region
```

```
    int splitNumEntities;
    int mergeNumEntities;

    Msocket *msocket;
    static IMBaseClient *gClient;

    IMBaseClient();
    ~IMBaseClient();

    virtual void tick();
    inline double getTime() { return simtime; }
    inline void setTime(double t) { simtime = t; }

    static int numrtt;
    static int getNextRTT() { numrtt++; return numrtt-1; }

    virtual void addOutEntity(IMBaseOutEntity *e) { outlist.insert(e); }
    virtual void remOutEntity(IMBaseOutEntity *e) { outlist.erase(e); }

    virtual void checkin(IMBaseOutEntity *,unsigned int,unsigned int);
    virtual void sendMoved(IMBaseOutEntity *e,McastPacket
                           &mpacket,unsigned int send_to
                           unsigned int to,unsigned int from,
                           short port=9976);

    virtual void sendPing(unsigned int send_to);
    virtual void sendMcast(unsigned int send_to,McastPacket &packet,
                           short port=9976);

    virtual void sendDelayedMcast(unsigned int send_to,
                                  McastPacket &packet,double delay);

    virtual void handleSplit(unsigned int address,
                             unsigned int newaddress[8]);
    virtual void handleMerge(unsigned int address);
    virtual void handleQuery(McastPacket &mpacket);

    virtual unsigned int OCTREE_search_server(IMBaseOutEntity *e);
    virtual unsigned int OCTREE_find_primary(IMBaseOutEntity *e,
                                            bool ask);
    virtual unsigned int OCTREE_ask_friends(IMBaseOutEntity *e,
                                           unsigned int address);
    virtual void OCTREE_Merge(IMNode *n);
    virtual void OCTREE_Split(IMNode *n);

    virtual bool MDHCP_getAddresses(int num,unsigned int *addrs);
};

#endif
```

2. IMBaseEntity.h

```
#ifndef _IMBASEENTITY
#define _IMBASEENTITY

#include <set.h>

#include "IMPacket.h"

#include "npsVec3f.h"
```

```
#include "IMCellRegion.h"

class IMBaseEntity
{
 public:

   unsigned int address; // Server assigned multicast address or guid.

   npsVec3f pos;         // position
   npsVec3f vel;         // velocity

   double lastUpdate;    // Time of last update (secs past Jan 1, 1970)

   CellRegion primary_region; //copy of primary region XXXX why?

 protected:

   npsVec3f oldpos;      // Position at time = lastUpdate

 public:

   enum { INBOUND,OUTBOUND };

   virtual int   mode() = 0;

   virtual int   runtime_type()     { return rtt; }
   virtual char *protocol_name()    { return module_name; }
   virtual char *protocol_url()     { return module_url; }
   virtual float protocol_version() { return module_version; }

   static int rtt;
   static char module_name[64];
   static char module_url[192];
   static float module_version;

   virtual void tick() = 0; // Update Entity

   virtual bool interestedIn(IMBaseEntity *e) { return false; }
   virtual void NotinterestedIn(IMBaseEntity *e) {}

   struct compare:public
   binary_function<IMBaseEntity *,IMBaseEntity *,bool>
   {
    bool operator()(IMBaseEntity *r1,IMBaseEntity *r2) const
              {
               return (r1->address < r2->address);
              }
   };
};

#endif
```

3. IMBaseInEntity.h

```
#ifndef _IMBASEINENTITY
#define _IMBASEINENTITY

#include "IMBaseEntity.h"
#include "IMPacket.h"
```

```
#include "string"
#include "map.h"

class IMBaseInEntity : public IMBaseEntity
{
 protected:

   // subscribed ref count, 0 == entity updates come from octree
   set<IMBaseEntity *,IMBaseEntity::compare> sub_set;

   double lastlastUpdate;
   npsVec3f oldoldpos;
   npsVec3f oldvel;
   float smooth_time;

   double dt;

   McastPacket lastPacket;

 public:

   IMBaseInEntity() {}

   IMBaseInEntity(McastPacket &mpacket)
   {
    address = mpacket.newaddress[1];
    update(mpacket);
   }

   virtual int mode() { return INBOUND; }

   virtual void tick(); // Update Entity
   virtual void update(McastPacket &mpacket);

   virtual int subscribed() { return (sub_set.size() > 0); }
   virtual bool subscribe(IMBaseEntity *e)
                         { return sub_set.insert(e).second;}
   virtual bool unsubscribe(IMBaseEntity *e)
                           { return sub_set.erase(e); }

   map<string,void *> attributeMap;
};

#endif
```

4. IMBaseOutEntity.h

```
#ifndef _IMBASEOUTENTITY
#define _IMBASEOUTENTITY

#include "npsVec3f.h"
#include "IMBaseEntity.h"
#include "IMBaseProtocol.h"
#include "IMCellRegion.h"

#include "map.h"

class IMNode;

class IMBaseOutEntity : public IMBaseEntity
```

```
{
 protected:

   float dead_thresh;       // Update if error > dead_thresh (m^2)
   float dead_timeout;      // Update if time > dead_timeout (s)

   void obtain_address();
   void checkin(unsigned int old);

   unsigned int find_primary();

   npsVec3f lastPos;

 public:

   //////////////////////////////////////////////
   // IE Parameters

   bool SubUnknownProtocols; // Should we by defualt subscribe to
                             // entities we did not have an IE for?

   float smallest_entity;     // Smallest entity we should look for.
                              // (Similar to smallest region calc)
   float smallest_entity_avg_bound_dia; // Used in smallest_entity calc.
   float smallest_entity_avg_max_vel;   // Used in smallest_entity calc.

   float roi;   //Tier 2 - Radius of interest
   float roi2;  //Tier 2 - Radius of interest squared

   // Outcome of filtering
   map<IMBaseEntity *,float,IMBaseEntity::compare> interestingEntities;


   //////////////////////////////////////////////
   // Smallest Region Parameters

   float smallest_region; //Smallest region this entity can fit in given
                          //current mode of movement

   float avg_bound_dia; // diameter used for smallest_region calc. (m)
   float avg_max_vel;   // Velocity used for smallest_region calc. (m/s)

   void AvgMaxVel(float v) { avg_max_vel = v; calc_smallest_region(); }
   void AvgBoundDia(float d){avg_bound_dia = d; calc_smallest_region();}

   set<IMNode *> current_nodes; // Set of interesting regions

 protected:

   void calc_smallest_region();
   double lastTick;                // Time this guy was ticked last
   double dt;                      // currenttime - lastTick

   void Update_dt();               // Updates dt;

   CellRegion region;   // Current location
   npsVec3f oldvel;  // Velocity at time = lastUpdate


 public:
   virtual int mode() { return OUTBOUND; }

   virtual void sendMoved(McastPacket &mpacket,
                          unsigned int send_to,unsigned int to,
                          unsigned int from, short port = 9976);

   virtual void forceSend(); // Used for pings
   virtual void tick();      // Update Entity

   IMBaseOutEntity(npsVec3f p, float search_r, float dia,
                   float vel,float timeout);
   IMBaseOutEntity(npsVec3f p, float search_r);

   ~IMBaseOutEntity();
};
#endif
```

5. IMBaseNode.h

```
#ifndef _IMBASENODE
#define _IMBASENODE

#include "set.h"

#include "IMCellRegion.h"

//Forward declare classes
class IMBaseOutEntity;

class IMNode
{
 public:
   CellRegion region; //Bounds
   IMNode *parent;
   IMNode *child[8];
   bool leaf;         //True if node has no children
   int ref;

   set<unsigned int> elist;

   IMNode(CellRegion r)
   {
    parent=NULL;
    leaf=true;
    for(int i=0;i<8;i++) child[i]=NULL;
    memcpy(&region,&r,sizeof(CellRegion));
    ref = 0;
   }

   IMNode()
   {
    parent=NULL;
    leaf=true;
    for(int i=0;i<8;i++) child[i]=NULL;
    ref = 0;
   }

   //Number on entities in child and parent
   //Returns -1 if all children present && not leaf nodes!
   virtual int childrenSize();
   virtual void splittree(unsigned int newaddr[8]);
   virtual void mergetree();

   virtual void mergeregions(CellRegion *region);
   virtual void fleshtree();
   virtual IMNode *findnode(unsigned int address);
   virtual bool remfromtree();
   virtual IMNode *addtotree(CellRegion r);
   virtual void   addtotree(void);
   virtual IMNode *XYZtonode(IMBaseOutEntity *e);
   virtual int searchXYZR(double x,double y,double z,double r,
                          CellRegion *cells,IMNode **nodes = NULL);
};

#endif
```

6. IMBaseProtocol.h

```
#ifndef __IMBASEPROTOCOL__
#define __IMBASEPROTOCOL__

#include "bbMappedClass.h"

#include "IMPacket.h"

// forward declare
class IMBaseInEntity;

class IMBaseProtocol : public bbMappedClass<IMBaseProtocol>
{
 public:

   IMBaseProtocol() :
bbMappedClass<IMBaseProtocol>("IMBaseClientModule") {setup();}
   IMBaseProtocol(const char *name) :
bbMappedClass<IMBaseProtocol>(name) {setup();}

   virtual void setup(void);

   virtual IMBaseInEntity *new_entity(McastPacket &mpacket);
   virtual void del_entity(IMBaseInEntity *);
};

class IMBaseProtocolIE
{
 public:
   IMBaseProtocol *protocol; //Which protocol is this for

   struct compare : public
   binary_function<IMBaseProtocolIE *,IMBaseProtocolIE *,bool>
   {
    bool operator()(IMBaseProtocolIE *r1,IMBaseProtocolIE *r2) const
                {
                 return (r1->protocol < r2->protocol);
                }
   };
};
#endif
```

7. IMCellRegion.h

```
#ifndef __IM_CELLREGION
#define __IM_CELLREGION

#include "npsVec3f.h"
#include "IMExtent.h"

struct CellRegion
{
  unsigned int address;
  Extent ext;
  int below;

  inline bool contains(npsVec3f p)
  {
   if (p[0] <= ext.x + ext.r && p[0] > ext.x - ext.r &&
       p[1] <= ext.y + ext.r && p[1] > ext.y - ext.r &&
       p[2] <= ext.z + ext.r && p[2] > ext.z - ext.r) return true;
   return false;
  }

  inline bool contains(npsVec3f p,float r)
  {
   r+=ext.r;
   if (p[0] <= ext.x + r && p[0] > ext.x - r &&
       p[1] <= ext.y + r && p[1] > ext.y - r &&
       p[2] <= ext.z + r && p[2] > ext.z - r) return true;
   return false;
  }
};

#endif
```

8. IMExtent.h

```
#ifndef __IM_EXTENT
#define __IM_EXTENT

// Messages from server
struct Extent
{
 double x;
 double y;
 double z;
 double r;
};

#endif
```

9. IMPacket.h

```
#ifndef __IM_PACKET
#define __IM_PACKET

#include "IMCellRegion.h"

// Messages to server
struct ServerPacket
```

```
{
 enum PacketType {SEARCH,CHANGE,MALLOC,LOCK,SPLIT,MERGE};
 enum PacketType type;
 union
 {
  struct
  {
   int num;
  } malloc;

  struct
  {
   unsigned int address;
   unsigned int newaddr[8];
  } split;

  struct
  {
   unsigned int address;
   double x;
   double y;
   double z;
   double r;
  } search;

  struct
  {
   unsigned int from;
   unsigned int to;
  } change;

 };
};

// Messages to mcast address
struct McastPacket
{
 enum PacketType {MOVED,PINGING,MERGE,SPLIT,QUERY,RESPOND};
 enum PacketType type;
 unsigned int address;
 union
     {
      unsigned int newaddress[8]; //XXXX is this used?!
      struct
           {
            unsigned int newaddress[2];
            double lastUpdate;
            double x;
            double y;
            double z;
            float vx;
            float vy;
            float vz;
            char module_name[64];
            char module_url[192];
            float module_version;
            char protocol_reserved[712];
           } moved; // 312 + 712 = 1024

      struct
           {
            unsigned int tcpaddress;
            unsigned int port;
            double r;
            double x;
            double y;
            double z;
           } query; // 40 bytes
     };
};

#endif
```

10. IMSocket.h

```
#ifndef _MSOCKET
#define _MSOCKET

#include <set.h>

#include "ace/SOCK_Dgram.h"

#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */

#include "ace/INET_Addr.h"

//A Socket that reads from multiple mcast addresses

class ACE_Export Msocket : public ACE_SOCK_Dgram
{
 public:
   Msocket(unsigned short port);
   ~Msocket(void);

   bool subscribe(unsigned int address);
   void unsubscribe(unsigned int address);
   int numsub() {return sub_list.size();}

 private:
   set<unsigned int> sub_list;
   in_addr ifaddr;
};

#endif
```

11. baseClient.c++

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#include <sys/time.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include "bbPrinter.h"
#include "bbThread.h"

#include "IMBaseClient.h"
#include "IMBaseOutEntity.h"
#include "IMBaseInEntity.h"
#include "IMBaseProtocol.h"

#include "IMSocket.h"
#include "ace/Handle_Set.h"

#include "IMPacket.h"
#include "IMBaseNode.h"

#define JOIN_DELAY 10

double smallest_region = 100000000.0;
int IMBaseClient::numrtt = 0;
IMBaseClient *IMBaseClient::gClient;

IMBaseClient::IMBaseClient()
{
 tree = NULL;
 internal_clock = true;
 auto_tick = true;
 k1 = 4.76; //constant for smallest region formula
 k2 = 0.01; //constant for dead reckoning formula
 MinTimeInRegion = 300.0; //Minimum time to spend in a region

 splitNumEntities = (2*k1*k2*MinTimeInRegion)/2;
 mergeNumEntities = 0.75 * splitNumEntities;

 printf("high water mark: %d\n",splitNumEntities);
 printf("low water mark: %d\n",mergeNumEntities);

 struct timeval tt;
printf("getting time...\n");
 gettimeofday(&tt);
printf("setting time...\n");
 setTime((double)tt.tv_sec + tt.tv_usec/1000000.0);
printf("Better not be this...\n");
 strcpy(server_name,DEFAULT_SERVER_NAME);
 server_port = DEFAULT_SERVER_PORT;
printf("Clearing lists...\n");
 outlist.clear();
 inlist.clear();
 setup();

printf("After set up.\n");
 //Turn on the base protocol. This is our default entity.
 base_proto = new IMBaseProtocol();
 IMBaseEntity::rtt = IMBaseClient::getNextRTT();

printf("Starting up network thread....\n");
 //Start up the network read thread
 readThread = new bbThread(readFunc,NULL);
printf("Leaving constructor....\n");
}

void IMBaseClient::setup()
{
```

```
printf("In setup!\n");
 struct hostent *en = gethostbyname(server_name);
 sa.sin_family=PF_INET;
 sa.sin_addr=*((struct in_addr *)(en->h_addr_list[0]));
 sa.sin_port=htons(server_port);

 // Set up the multicast socket
 int on=1,ttl = 15;
 unsigned char off = 0;
 msocket = new Msocket(9976);
 s2 = socket(AF_INET, SOCK_DGRAM, 0);
 setsockopt(s2, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
 if (setsockopt(s2, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on)) < 0)
    perror("setsockopt SO_REUSEPORT");
 if (setsockopt(s2, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
    perror("setsockopt SO_REUSEADDR");
 if (setsockopt(s2, IPPROTO_IP, IP_MULTICAST_LOOP, &off, sizeof(off)) <
0)
    perror("setsockopt IP_MULTICAST_LOOP");

 // Set up our tcp socket
 struct in_addr bind_address;
 bind_address.s_addr = htonl(INADDR_ANY);

 struct sockaddr_in sa_client;
 sa_client.sin_family=AF_INET;
 sa_client.sin_addr=bind_address;
 sa_client.sin_port=htons(DEFAULT_CLIENT_PORT);

 ourtcpsocket = socket(PF_INET,SOCK_STREAM,0);
 if (bind(ourtcpsocket,(struct sockaddr *)&sa_client,
                      sizeof(struct sockaddr_in)) == -1) perror("bind");

 listen(ourtcpsocket,1000);

 en = gethostbyname(server_name);
 clientaddress = *((int *)(en->h_addr_list[0]));

 struct in_addr grpaddr;
 grpaddr.s_addr = clientaddress;
 printf("clientaddress = %s\n",inet_ntoa(grpaddr));
}


IMBaseClient::~IMBaseClient()
{
 printf("IMBaseClient shutting down\n");
 delete readThread;
 printf("Killed readThread.\n");
 delete msocket;
 printf("deleted msocket.\n");
 printf("IMBaseClient shutdown\n");
}

void IMBaseClient::tick()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 struct timeval tt;
 gettimeofday(&tt);
 double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
 // Go through wait respond queue
 // If something is ready, it means that we must check the server
 multimap<double,IMBaseOutEntity *>::iterator k = delayRespond.begin();
 while(k != delayRespond.end() && (*k),first < tod)
      {
printf("Ask friends failed... must ask server!\n");
       OCTREE_search_server((*k).second);
       delayRespond.erase(k);
       k = delayRespond.begin();
      }

 // Go through delay send queue
 multimap<double,packet_type>::iterator p = delaySend.begin();
 while(p != delaySend.end() && (*p).first < tod)
      {
       sendMcast((*p).second.first,(*p).second.second);
       delaySend.erase(p);
       p = delaySend.begin();
      }

 // Go through delay split check queue
 multimap<double,unsigned int>::iterator q = delaySplit.begin();
 while(q != delaySplit.end() && (*q).first < tod)
      {
       // We erase first because 'OCTREE_Split' can mess with the queue
       unsigned int node_addr = (*q).second;
       delaySplit.erase(q);
       q = delaySplit.begin();

       IMNode *nn;
       if ((nn = tree->findnode(node_addr)) != NULL)
          if (nn->leaf && nn->elist.size() > splitNumEntities)
             OCTREE_Split(nn);
      }

 // Go through delay merge check queue
 multimap<double,unsigned int>::iterator r = delayMerge.begin();
 while(r != delayMerge.end() && (*r).first < tod)
      {
       // We erase first because 'OCTREE_Merge' can mess with the queue
       unsigned int node_addr = (*r).second;
       delayMerge.erase(r);
       r = delayMerge.begin();

       IMNode *nn;
       if ((nn = tree->findnode(node_addr)) != NULL)
          if (!nn->leaf && nn->childrenSize() >= 0 &&
              nn->childrenSize() < mergeNumEntities)
             OCTREE_Merge(nn);
      }

 //Set simtime based on tod
 if (internal_clock) setTime(tod);

 set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i;
 set<IMBaseInEntity *,IMBaseEntity::compare>::iterator j;

 //XXXX should we just erase the list and start over every time?
 // This manages the interestingEntities list for inbounds.
 for(j = inlist.begin();j!=inlist.end();j++)
```

```
    for(i = outlist.begin();i!=outlist.end();i++)
       {
        float dist2 = ((*i)->pos - (*j)->pos).lengthSqr();
        if (dist2 < (*i)->roi2 && (*i)->interestedIn(*j))
           (*i)->interestingEntities.insert
                 (pair<IMBaseEntity *,float>(*j,dist2));
        else
           {
           (*i)->interestingEntities.erase(*j);
           (*i)->NotinterestedIn(*j);
           }
       }

 // This manages the interestingEntities list for outbounds.
 for(i = outlist.begin();i!=outlist.end();i++)
    {
     if ((*i)->smallest_region < smallest_region)
        smallest_region = (*i)->smallest_region;
     set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i2 = i;
     for(i2++;i2!=outlist.end();i2++)
        {
         float dist2 = ((*i)->pos - (*i2)->pos).lengthSqr();
         if (dist2 < (*i)->roi2 && (*i)->interestedIn(*i2))
            (*i)->interestingEntities.insert
                  (pair<IMBaseEntity *,float>(*i2,dist2));
         else
            (*i)->interestingEntities.erase(*i2);
         if (dist2 < (*i2)->roi2 && (*i2)->interestedIn(*i))
            (*i2)->interestingEntities.insert
                   (pair<IMBaseEntity *,float>(*i,dist2));
         else
            (*i2)->interestingEntities.erase(*i);
        }
    }

 if (auto_tick)
    {
     for(i = outlist.begin();i!=outlist.end();i++)
        (*i)->tick();

     for(j = inlist.begin();j!=inlist.end();j++)
        (*j)->tick();
    }
}

void IMBaseClient::
checkin(IMBaseOutEntity *e,unsigned int from,unsigned int to)
{
 static McastPacket p;
 e->sendMoved(p,from,to,from);
 e->sendMoved(p,to,to,from);
}

void IMBaseClient::
sendDelayedMcast(unsigned int send_to,McastPacket &packet,double delay)
{
 struct timeval tt;
 gettimeofday(&tt);
 double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
 packet_type packetPair(send_to,packet);
 multimap<double,packet_type>::value_type
```

```
          valuePair(delay+tod,packetPair);
 delaySend.insert(valuePair);
}

void IMBaseClient::
sendMcast(unsigned int send_to,McastPacket &packet,short port)
{
 struct sockaddr_in sa_mcast;
 sa_mcast.sin_family=AF_INET;
 struct in_addr grpaddr;
 grpaddr.s_addr = send_to;
 sa_mcast.sin_addr=grpaddr;
 sa_mcast.sin_port=htons(port);

 sendto(s2,&packet,sizeof(McastPacket),0,&sa_mcast,sizeof(sa_mcast));
}

void IMBaseClient::
sendPing(unsigned int send_to)
{
 McastPacket packet;
 packet.type=McastPacket::PINGING;
 packet.address=send_to;

 //Send out ping in 30 seconds.
 sendDelayedMcast(send_to,packet,JOIN_DELAY);
}

void IMBaseClient::OCTREE_Merge(IMNode *n)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 struct in_addr grpaddr;
 grpaddr.s_addr = n->region.address;
 printf("Merge %s?\n",inet_ntoa(grpaddr));

 ServerPacket packet;

 packet.type = ServerPacket::LOCK;
 packet.split.address = n->region.address;

 int lock=0;

 printf("Lock %s?\n",inet_ntoa(grpaddr));
 int s = socket(PF_INET,SOCK_STREAM,0);
 if ((connect(s,&sa,sizeof(struct sockaddr_in)))==-1) perror("connect");
 write(s,&packet,sizeof(ServerPacket));
 read(s,&lock,sizeof(int));
 close(s);

 if (lock)
    {
     printf("Locked %s\n",inet_ntoa(grpaddr));
     printf("Merging %s on server\n",inet_ntoa(grpaddr));
     packet.type = ServerPacket::MERGE;
     packet.split.address = n->region.address;
     s = socket(PF_INET,SOCK_STREAM,0);
     if ((connect(s,&sa,sizeof(struct sockaddr_in)))==-1)
        perror("connect");
     write(s,&packet,sizeof(ServerPacket));
     read(s,&lock,sizeof(int));
     close(s);

     if (lock)
        {
         printf("Merging %s on mcast\n",inet_ntoa(grpaddr));
         McastPacket mpacket;
         mpacket.type=McastPacket::MERGE;
         mpacket.address=n->region.address;
         for(int i=0;i<8;i++)
            {
             sendMcast(n->child[i]->region.address,mpacket);
             sendDelayedMcast(n->child[i]->region.address,
                              mpacket,JOIN_DELAY);
            }
         handleMerge(n->region.address);
         return;
        }
     else
        {
         printf("Merge failed must have trucated tree tree.\n");

         //artifically make child[0] a branch.
         n->child[0]->leaf = true;
        }
    }

 // Lock was bad. Something is most likly wrong,
 // remove near-future request
 multimap<double,unsigned int>::iterator q = delayMerge.begin();
 for(;q != delayMerge.end();q++)
    if ( (*q).second == n->region.address )
       {
        delayMerge.erase(q);
        q = delayMerge.begin();
       }
}

void IMBaseClient::OCTREE_Split(IMNode *n)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 struct in_addr grpaddr;
 grpaddr.s_addr = n->region.address;
 printf("Split %s?\n",inet_ntoa(grpaddr));

 ServerPacket packet;

 packet.type = ServerPacket::LOCK;
 packet.split.address = n->region.address;

 int lock=0;

 printf("Lock %s?\n",inet_ntoa(grpaddr));
 int s = socket(PF_INET,SOCK_STREAM,0);
 if ((connect(s,&sa,sizeof(struct sockaddr_in)))==-1) perror("connect");
 write(s,&packet,sizeof(ServerPacket));
 read(s,&lock,sizeof(int));
 close(s);

 if (lock)
    {
     printf("Locked %s\n",inet_ntoa(grpaddr));
     unsigned int addrs[8];
     if (!MDHCP_getAddresses(8,addrs))
         {
          //Tell server never mind...
         }
     else
         {
          printf("Splitting %s on server\n",inet_ntoa(grpaddr));
          packet.type = ServerPacket::SPLIT;
          packet.split.address = n->region.address;
          for(int i=0;i<8;i++) packet.split.newaddr[i] = addrs[i];
          s = socket(PF_INET,SOCK_STREAM,0);
          if ((connect(s,&sa,sizeof(struct sockaddr_in)))==-1)
             perror("connect");
          write(s,&packet,sizeof(ServerPacket));
          read(s,&lock,sizeof(int));
          close(s);

          if (lock)
             {
              printf("Splitting %s on mcast\n",inet_ntoa(grpaddr));
              McastPacket mpacket;
              mpacket.type=McastPacket::SPLIT;
              mpacket.address=n->region.address;
              for(i=0;i<8;i++) mpacket.newaddress[i]=addrs[i];
              sendMcast(n->region.address,mpacket);
              sendDelayedMcast(n->region.address,mpacket,JOIN_DELAY);
              handleSplit(n->region.address,addrs);
              return;
             }
         }
    }

 // Lock was bad. Something is mostlikly wrong, remove near-future
request
 multimap<double,unsigned int>::iterator q = delaySplit.begin();
 for(;q != delaySplit.end();q++)
    if ( (*q).second == n->region.address )
       {
        delaySplit.erase(q);
        q = delaySplit.begin();
       }
}

void IMBaseClient::
sendMoved(IMBaseOutEntity *e,McastPacket &mpacket,unsigned int send_to,
          unsigned int to,unsigned int from, short port)
{
 if (to != from)
    {
     IMNode *n = tree->findnode(from);
     if (n)
        if (n->elist.erase(mpacket.newaddress[1]))
           {
            if (n->leaf) n=n->parent;
            if (!n->leaf && n->childrenSize() >= 0 &&
                 n->childrenSize() < mergeNumEntities)
               {
                struct timeval tt;
                gettimeofday(&tt);
                double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
                double delay = 20.0 + JOIN_DELAY + rand()%
                              (n->childrenSize()?n->childrenSize():1);
                multimap<double,unsigned int>::value_type
                        valuePair(delay+tod,n->region.address);
                delayMerge.insert(valuePair);
               }
           }
     n = tree->findnode(to);
     if (n)
        if (n->elist.insert(mpacket.newaddress[1]).second)
           {
            if (n->leaf && n->elist.size() > splitNumEntities &&
               (n->region.ext.r/2.0) > e->smallest_region)
               {
                struct timeval tt;
                gettimeofday(&tt);
                double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
                double delay = 20.0 + JOIN_DELAY +
                               rand()%n->elist.size();
                multimap<double,unsigned int>::value_type
                        valuePair(delay+tod,to);
                delaySplit.insert(valuePair);
               }
           }

    }

 sendMcast(send_to,mpacket,port);
}

//Allocate num multicast addresses from MDHCP server.
//XXXX Currently this is handled by IM server!
bool IMBaseClient::MDHCP_getAddresses(int num,unsigned int *addrs)
{
 ServerPacket packet;

 packet.type = ServerPacket::MALLOC;
 packet.malloc.num = num;

 int s = socket(PF_INET,SOCK_STREAM,0);
 if ((connect(s,&sa,sizeof(struct sockaddr_in)))==-1) perror("connect");
 write(s,&packet,sizeof(ServerPacket));
 read(s,addrs,sizeof(unsigned int)*num);
 close(s);

 for(int i=0;i<num;i++) if (addrs[i] == 0) return false;
 return true;
}

unsigned int IMBaseClient::
OCTREE_ask_friends(IMBaseOutEntity *e,unsigned int address)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 McastPacket mpacket;
 mpacket.type = McastPacket::QUERY;
 mpacket.address = address;
 mpacket.query.tcpaddress = clientaddress;
 mpacket.query.port = DEFAULT_CLIENT_PORT;
 mpacket.query.x = e->pos[0];
 mpacket.query.y = e->pos[1];
 mpacket.query.z = e->pos[2];
 mpacket.query.r = e->roi;
 sendMcast(address,mpacket);

 pair<double,IMBaseOutEntity *> p;
 p.first=30;
 p.second = e;
 delayRespond.insert(p);

 return 0;
}

unsigned int IMBaseClient::OCTREE_search_server(IMBaseOutEntity *e)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 static CellRegion region[MAX_READ_REGIONS];
 int s;

 int n;
 ServerPacket packet;

 packet.type = ServerPacket::SEARCH;
 packet.search.address = e->primary_region.address;
 packet.search.r = e->roi;
 packet.search.x = e->pos[0];
 packet.search.y = e->pos[1];
 packet.search.z = e->pos[2];
 s = socket(PF_INET,SOCK_STREAM,0);

 if ((connect(s,&sa,sizeof(struct sockaddr_in)))==-1) perror("connect");
 write(s,&packet,sizeof(ServerPacket));
 read(s,&n,sizeof(int));
 bigread(s,region,sizeof(CellRegion)*n);
 close(s);
 printf("Got %d regions\n",n);

 if (tree == NULL) tree = new IMNode(region[0]);
 tree->mergeregions(region,n);
 e->primary_region.address = 0;
 e->primary_region.ext.r = 0;
 return OCTREE_find_primary(e,true);
}

// Finds entities primary region
unsigned int IMBaseClient::
OCTREE_find_primary(IMBaseOutEntity *e,bool ask)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 IMNode *tmp;

 //Check to see if we are currently in the correct region.
 //Handles splits, and smallest regions....
 if (e->primary_region.contains(e->pos))
    {
     tmp = tree->findnode(e->primary_region.address);
     if (tmp)
        {
         if (tmp->leaf)
```

```
            return 0;
         else
            {
             if (e->smallest_region > tmp->region.ext.r/2.0)
                return 0;
             else
                tmp = tmp->XYZtonode(e);
            }
        }
    }
 else // Search the tree in memory
    tmp = tree->XYZtonode(e);

 if (!tmp) tmp = tree;
 if (tmp) // If we found an answer
    {
     if (tmp->region.address != 0 && tmp != tree) // Is it a real region
        {
         unsigned int rval = e->primary_region.address;
         memcpy(&e->primary_region,&tmp->region,sizeof(CellRegion));
         return rval;
        }
     else // It's a fake! (or root of tree...)
        {
         if (tmp->parent) tmp=tmp->parent;
         else { printf("entity primary region in root!\n"); exit(0); }

         for(int f=0;f<8;f++)
            {
             if (tmp->child[f] && tmp->child[f]->region.address != 0)
                {
                 if (ask)
                    return OCTREE_ask_friends(e,
                                       tmp->child[f]->region.address);
                 else
                    return 0;
                }
            }
        }
    }

 printf("You should never see this!!!\n");
 return 0;
}

void IMBaseClient::filter(McastPacket &mpacket,IMBaseInEntity *ng)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 //Does Protocol Exist? O(lg P)
 IMBaseProtocol *protocol = IMBaseProtocol::
                            findObject(mpacket.moved.module_name);
 if (protocol == NULL)
    {
     printf ("'%s' not loaded... tring!\n",mpacket.moved.module_name);
     bbModule::load(mpacket.moved.module_name,
             mpacket.moved.module_version,0,mpacket.moved.module_url);
     protocol = IMBaseProtocol::findObject(mpacket.moved.module_name);

     if (protocol == NULL)
        {
```

```
        bbError << "Could not load protocol module `"
                << mpacket.moved.module_name << "`!" << endl;
        protocol = base_proto;
       }
   }

//Does Entity Exist?
set<IMBaseInEntity *,IMBaseEntity::compare>::iterator i;

// Yes - an old guy. O(lg N)
if ((i = inlist.find(ng)) != inlist.end())
   {
    if ((*i)->lastUpdate < mpacket.moved.lastUpdate)
       (*i)->update(mpacket);
   }
else // No - a new guy O(lg N)
   {
    i = inlist.insert(protocol->new_entity(mpacket)).first;
   }

// If he is checking out, delete him!
if (mpacket.address == 0)
   {
    printf("Killing entity...\n");
    IMBaseInEntity *tmp = *i;
    inlist.erase(i);
    protocol->del_entity(tmp);
   }
else
//Keep track of number and who is in each region.
//Atempt to split or merge if needed.
if ((*i)->primary_region.address != mpacket.address)
   {
    IMNode *n = tree->findnode((*i)->primary_region.address);
    if (n)
       if (n->elist.erase(mpacket.newaddress[1]))
          {
           if (n->leaf) n=n->parent;
           if (!n->leaf && n->childrenSize() >= 0 &&
                n->childrenSize() < mergeNumEntities)
              {
               struct timeval tt;
               gettimeofday(&tt);
               double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
               double delay = 20.0 + JOIN_DELAY + rand()%
                             (n->childrenSize()?n->childrenSize():1);
               multimap<double,unsigned int>::value_type
                       valuePair(delay+tod,n->region.address);
               delayMerge.insert(valuePair);
              }
          }
    n = tree->findnode(mpacket.address);
    if (n)
       if (n->elist.insert(mpacket.newaddress[1]).second)
          {
           if (n->leaf && n->elist.size() > splitNumEntities &&
              (n->region.ext.r/2.0) > smallest_region)
              {
               struct timeval tt;
               gettimeofday(&tt);
               double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
```

```
                 double delay = 20.0 + JOIN_DELAY + rand()%
                                n->elist.size();
                 multimap<double,unsigned int>::value_type
                          valuePair(delay+tod,mpacket.address);
                 delaySplit.insert(valuePair);
                }
            }
    }

}


void IMBaseClient::readFunc(bbThread *,bbData *that)
{
 //XXXX avoid race condition
 if (IMBaseClient::gClient != NULL)
    IMBaseClient::gClient->mcastcheck();
 else
    sleep(0);
}

void IMBaseClient::mcastcheck()
{
 int rval;
 McastPacket mpacket;
 ACE_Handle_Set handle_set;

 handle_set.reset();
 handle_set.set_bit(msocket->get_handle());
 handle_set.set_bit(ourtcpsocket);

 // Time out so thread will unblock.
 ACE_Time_Value t2(1,0);

 if ((rval = ACE_OS::select((ourtcpsocket)+1,handle_set,
      NULL,NULL,&t2)) != 0)
    {
     if (rval == -1) perror("select");
     else
        if (handle_set.is_set(ourtcpsocket))
           {
            printf("Got a reply to our query!\n");
            int n;
            struct sockaddr sa_client;
            int cs = sizeof(sa_client);
            int s = accept(ourtcpsocket,&sa_client,&cs);
            static CellRegion cells[MAX_READ_REGIONS];
            read(s,&n,sizeof(int));
            bigread(s,cells,sizeof(CellRegion)*n);
            close(s);
            printf("Got %d regions\n",n);
            tree->mergeregions(cells,n);
            // run though delayRespond queue to see if we can
            // take anytthing out!
            multimap<double,IMBaseOutEntity *>::iterator
                     k = delayRespond.begin();
            ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);
            while(k != delayRespond.end())
                 {
                  // This message fixed his problem
                  if (OCTREE_find_primary((*k).second,false))
```

```
                     {
                      multimap<double,IMBaseOutEntity *>::iterator
                      i = k;
                      i++;
                      delayRespond.erase(k);
                      k = i;
                     }
                 }
            }
        else
           {
            ACE_OS::recvfrom(msocket->get_handle(),(char *)&mpacket,
                             sizeof(McastPacket), 0,0,0);
            switch (mpacket.type)
                   {
                    case McastPacket::MOVED:
                             {
                              static IMBaseInEntity *ng =
                                     new IMBaseInEntity(mpacket);
                              ng->address = mpacket.newaddress[1];
                              filter(mpacket,ng);
                             }
                             break;

                    case McastPacket::PINGING:
                    {
                     ACE_Guard<bbMutex>
                               guard(IMBaseClient::gClient->tree_mutex);
                     set<IMBaseOutEntity *,IMBaseEntity::compare>
                               ::iterator i;
                     for(i = outlist.begin();i!=outlist.end();i++)
                     if((*i)->primary_region.address == mpacket.address)
                       (*i)->forceSend();

// Remove ping request if we already go one.
 multimap<double,packet_type>::iterator p = delaySend.begin();
 for(;p != delaySend.end(); p++)
    {
     if ((*p).second.first == mpacket.address &&
         (*p).second.second.type == McastPacket::PINGING)
        {
         delaySend.erase(p);
         printf("Ping request removed..\n");
         break;
        }
    }

                             }
                             break;

                    case McastPacket::MERGE:
                             {
                              struct in_addr grpaddr;
                              grpaddr.s_addr = mpacket.address;
                              handleMerge(mpacket.address);
                             }
                             break;

                    case McastPacket::SPLIT:
                       {
                        struct in_addr grpaddr;
                        grpaddr.s_addr = mpacket.address;
                        handleSplit(mpacket.address,mpacket.newaddress);
                       }
                       break;

                    case McastPacket::QUERY:
                             {
                              handleQuery(mpacket);
                             }
                             break;

                    default: printf("Unknown message from socket!\n");
                   }
           }
    }
}

// Joe blow wants a region intersection.
void IMBaseClient::handleQuery(McastPacket &mp)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);
 printf("We got a Query!\n");

 static CellRegion cells[MAX_READ_REGIONS];

 int num = tree->searchXYZR(mp.query.x,mp.query.y,mp.query.z,
                            mp.query.r,cells);

 if (num == 0) return; // We don't know the answer

 //Connect to client needing help via tcp
 struct sockaddr_in tmpsa;
 struct hostent *en = gethostbyaddr(&mp.query.tcpaddress,4,AF_INET);
 tmpsa.sin_family=PF_INET;
 tmpsa.sin_addr=*((struct in_addr *)(en->h_addr_list[0]));
 tmpsa.sin_port=htons((short)mp.query.port);

 int s = socket(PF_INET,SOCK_STREAM,0);
 if ((connect(s,&tmpsa,sizeof(struct sockaddr_in)))==-1)
     perror("connect");
 write(s,&num,sizeof(int));
 bigwrite(s,cells,sizeof(CellRegion)*num);
 close(s);

 printf("Sent them back %d regions!\n",num);
}

void IMBaseClient::handleMerge(unsigned int address)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 IMNode *parent = tree->findnode(address);
 if (!parent) return;
 if (parent->leaf) return;
 for(int x=0;x<8;x++)
    if (parent->child[x])
       if (!parent->child[x]->leaf) return;

 printf("Address %d has been merged!\n",address);

 set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i
 for(I = outlist.begin();i!=outlist.end();i++)
    for(x=0;x<8;x++)
       if (parent->child[x])
          if ((*i)->primary_region.address ==
              parent->child[x]->region.address)
             {
              memcpy(&(*i)->primary_region,&parent->region,
                     sizeof(CellRegion));
              checkin((*i),parent->child[x]->region.address,
                      (*i)->primary_region.address);
             }

 parent->mergetree();

 parent->elist.clear();
}

void IMBaseClient::
handleSplit(unsigned int address,unsigned int newaddress[8])
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 IMNode *parent = tree->findnode(address);
 if (!parent) return;
 if (!parent->leaf) return;

 printf("Address %d has been split!(%d)\n",address,parent->leaf);

 parent->splittree(newaddress);

 parent->elist.clear();

 set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i
 for(i = outlist.begin();i!=outlist.end();i++)
    if ((*i)->primary_region.address == address)
       {
        OCTREE_find_primary(*i,true);
        if ((*i)->primary_region.address != address)
           {
            checkin((*i),address,(*i)->primary_region.address);
           }
       }
}
```

12. baseInEntity.c++

```
#include "IMBaseEntity.h"

int IMBaseEntity::rtt;
char IMBaseEntity::module_name[64] = "IMBaseClientModule";
char IMBaseEntity::module_url[192] = "";
float IMBaseEntity::module_version = 0.894;

#include "IMBaseClient.h"
#include "IMBaseInEntity.h"

void IMBaseInEntity::update(McastPacket &mpacket)
{
 memcpy(&lastPacket,&mpacket,sizeof(McastPacket));
 npsVec3f newpos(mpacket.moved.x,mpacket.moved.y,mpacket.moved.z);
 npsVec3f newvel(mpacket.moved.vx,mpacket.moved.vy,mpacket.moved.vz);

 oldoldpos = oldpos;
 oldpos = newpos;

 oldvel = vel;
 vel = newvel;

 lastlastUpdate = lastUpdate;
 lastUpdate = mpacket.moved.lastUpdate;
}

void IMBaseInEntity::tick()
{
 dt = IMBaseClient::gClient->getTime()-lastUpdate;
 pos = oldpos + vel * dt;

 if (dt < smooth_time)
    {
     double dt2 = IMBaseClient::gClient->getTime()-lastlastUpdate;
     float frac = dt/smooth_time;
     pos = pos * frac + (oldoldpos + oldvel * dt2) * (1.0f-frac);
    }
}
```

13. baseNode.c++

```
#include "IMSocket.h"
#include "IMBaseNode.h"
#include "IMBaseClient.h"
#include "IMBaseInEntity.h"
#include "IMBaseOutEntity.h"

void IMNode::mergeregions(CellRegion *regions)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 int i = 0;
 IMNode *p;

 p = findnode(regions[0].address);
 if (p != NULL) p->region.below = regions[0].below;
 else printf("Error.. new tree begins outside old\n");

 while(p != NULL && i < n-1)
      {
       i++;
       IMNode *c = p->findnode(regions[i].address);

       if (c == NULL) c = p->addtotree(regions[i]);
       c->region.below = regions[i].below;
       p->region.below -= (c->region.below + 1);

       if (c->region.below) p = c;
       while(p != NULL && p->region.below == 0) p = p->parent;
      }

 fleshtree();
}
```

```
//Fills children with '0' regions
//If entity is interested in '0' region, it means that
//he needs more octree info
void IMNode::fleshtree()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 unsigned char mask = 0x0;
 int i;

 for(i=0;i<8;i++)
    if (child[i])
       {
        child[i]->fleshtree();
        mask |=(0x1<<(((char)(child[i]->region.ext.x < region.ext.x))<<2
                    | ((char)(child[i]->region.ext.y < region.ext.y))<<1
                    | ((char)(child[i]->region.ext.z < region.ext.z))));
       }

 if (mask != 0x0 && mask != 0xff)
    {
     for(int x=0; x < 8; x++)
        {
         if ((0x1<<x) & mask) continue;

         CellRegion r;
         r.address = 0;
         r.ext.r = region.ext.r/2.0;
         r.ext.x = 0.0;
         r.ext.y = 0.0;
         r.ext.z = 0.0;
         for(i=0;i<8;i++)
            {
             if (child[i] == NULL)
                {
                 child[i] = new IMNode(r);
                 child[i]->parent = this;
                 child[i]->region.ext.x = region.ext.x +
                          r.ext.r*((0x4 & x)?-1:1);
                 child[i]->region.ext.y = region.ext.y +
                          r.ext.r*((0x2 & x)?-1:1);
                 child[i]->region.ext.z = region.ext.z +
                          r.ext.r*((0x1 & x)?-1:1);
                 break;
                }
            }
        }
    }
}

IMNode *IMNode::findnode(unsigned int address)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 if (region.address == address) return this;
 for(int i=0; i<8; i++)
    {.
     if (child[i])
        {
         struct IMNode *rval = child[i]->findnode(address);
         if (rval != NULL) return rval;
        }
    }
 return NULL;
}

int IMNode::childrenSize()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 int res = 0;
 for(int i=0;i<8;i++)
    {
     if (child[i])
        {
         if (!child[i]->leaf) return -1;
         res += child[i]->elist.size();
        }
     else return -1;
    }
 return res;
}

void IMNode::splittree(unsigned int newaddr[8])
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 CellRegion r;
 r.ext.r = region.ext.r/2.0;

 r.ext.x = region.ext.x - r.ext.r;
 r.ext.y = region.ext.y - r.ext.r;
 r.ext.z = region.ext.z - r.ext.r;
 r.address = newaddr[0];
 addtotree(r);

 r.ext.x = region.ext.x - r.ext.r;
 r.ext.y = region.ext.y - r.ext.r;
 r.ext.z = region.ext.z + r.ext.r;
 r.address = newaddr[1];
 addtotree(r);

 r.ext.x = region.ext.x - r.ext.r;
 r.ext.y = region.ext.y + r.ext.r;
 r.ext.z = region.ext.z - r.ext.r;
 r.address = newaddr[2];
 addtotree(r);

 r.ext.x = region.ext.x - r.ext.r;
 r.ext.y = region.ext.y + r.ext.r;
 r.ext.z = region.ext.z + r.ext.r;
 r.address = newaddr[3];
 addtotree(r);

 r.ext.x = region.ext.x + r.ext.r;
 r.ext.y = region.ext.y - r.ext.r;
 r.ext.z = region.ext.z - r.ext.r;
 r.address = newaddr[4];
 addtotree(r);

 r.ext.x = region.ext.x + r.ext.r;
 r.ext.y = region.ext.y - r.ext.r;
 r.ext.z = region.ext.z + r.ext.r;
 r.address = newaddr[5];
 addtotree(r);

 r.ext.x = region.ext.x + r.ext.r;
 r.ext.y = region.ext.y + r.ext.r;
 r.ext.z = region.ext.z - r.ext.r;
 r.address = newaddr[6];
 addtotree(r);

 r.ext.x = region.ext.x + r.ext.r;
 r.ext.y = region.ext.y + r.ext.r;
 r.ext.z = region.ext.z + r.ext.r;
 r.address = newaddr[7];
 addtotree(r);
}

void IMNode::mergetree()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 leaf = true;
 for (int i =0;i<8;i++)
     if (child[i])
        {
         //XXX is there a quicker way!?
         set<IMBaseOutEntity *>::iterator
         ee = IMBaseClient::gClient->outlist.begin();
         for(;ee != IMBaseClient::gClient->outlist.end(); ee++)
            (*ee)->current_nodes.erase(child[i]);

         delete child[i];
         child[i] = 0;
        }
}

bool IMNode::remfromtree()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 ref--;

 if (ref > 0) return false;

 IMBaseClient::gClient->msocket->unsubscribe(region.address);

 set<unsigned int>::iterator i;
 for(i = elist.begin();i != elist.end(); i++)
    {
     IMBaseInEntity ng;
     ng.address = *i;
     set<IMBaseInEntity *>::iterator i =
         IMBaseClient::gClient->inlist.find(&ng);
     if (i != IMBaseClient::gClient->inlist.end())
        {
         IMBaseClient::gClient->inlist.erase(i);
         if (!(*i)->subscribed()) delete *i;
        }
    }

 // Replace us with a empty node.
 region.address = 0;

 return true;
}

void IMNode::addtotree()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 ref++;
}

IMNode *IMNode::addtotree(CellRegion r)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 // See if it is already there
 for(int x=0;x<8;x++)
    {
     if (child[x] != NULL && child[x]->region.address == r.address)
       return child[x];
    }

 // See if we can replace an empty node
 for(x=0;x<8;x++)
    if (child[x] != NULL)
       {
    if (child[x]->region.ext.x == r.ext.x &&
        child[x]->region.ext.y == r.ext.y &&
        child[x]->region.ext.z == r.ext.z)
       {
    if (child[x]->region.address == 0)
       {
        leaf = false;
        child[x]->region.address = r.address;
      if (IMBaseClient::gClient->msocket->
          subscribe(child[x]->region.address))
           IMBaseClient::gClient->sendPing(child[x]->region.address);
        return child[x];
       }
    else
       {
        leaf = false;
        child[x]->region.address = r.address;
        if (IMBaseClient::gClient->msocket->
           subscribe(child[x]->region.address))
           IMBaseClient::gClient->sendPing(child[x]->region.address);
        return child[x];
       }
       }
       }


 // If not, see if we can just stick it in the tree
 if (x==8)
 for(x=0;x<8;x++)
    if (child[x] == NULL)
       {
        child[x] = new IMNode(r);
        child[x]->parent = this;
        leaf = false;
        if (IMBaseClient::gClient->msocket->
            subscribe(child[x]->region.address))
           IMBaseClient::gClient->sendPing(child[x]->region.address);
        return child[x];
       }

 // This should never happen!
 printf("Error! No room in tree!!! (this should never happen!)\n");
 return NULL;
}


IMNode *IMNode::XYZtonode(IMBaseOutEntity *e)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 if (region.contains(e->pos) && e->smallest_region < region.ext.r)
    {
     for(int i=0;i<8;i++)
        {
         if (child[i])
            {
             IMNode *rval = child[i]->XYZtonode(e);
             if (rval) return rval;
            }
        }
     return this;
    }
 return NULL;
}

int IMNode::searchXYZR(double x,double y,double z,double r,CellRegion
*cells,IMNode **nodes)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 //Always return parent.
 CellRegion *origregs = cells;
 cells[0].ext.x = region.ext.x;
 cells[0].ext.y = region.ext.y;
 cells[0].ext.z = region.ext.z;
 cells[0].ext.r = region.ext.r;
 cells[0].address = region.address;
 cells = &cells[1];

 if (nodes)
    {
     nodes[0] = this;
     nodes = &nodes[1];
    }
 int count = 1;

 // Out of the eight child, which do we intersect?
 for(int i=0;i<8;i++)
    {
     if (child[i])
        {
         if (child[i]->region.contains(npsVec3f(x,y,z),r))
            {
             int c = child[i]->searchXYZR(x,y,z,r,cells,nodes);
             cells = &cells[c];
```

```
              if (nodes) nodes = &nodes[c];
              count += c;
             }
         }
     }

 //Set number of regions below this one.
 origregs->below = count-1;

 //Return the number of regions.
 return count;
}
```

14. baseOutEntity.c++

```
#include "IMBaseClient.h"
#include "IMBaseOutEntity.h"
#include "IMBaseNode.h"

#include "vector.h"

IMBaseOutEntity::IMBaseOutEntity(npsVec3f p, float search_r, float dia,
                                  float vel, float timeout)
{
 roi = search_r;
 roi2 = roi*roi;
 primary_region.address = 0;
 address = 0;

 AvgBoundDia(dia);
 AvgMaxVel(vel);

 dead_timeout = timeout;
 oldvel.makeNull();
 oldpos = pos = p;
 oldvel[0] = 0.0;  oldvel[1] = 0.0;  oldvel[2] = 0.0;
 obtain_address();
 IMBaseClient::gClient->addOutEntity(this);
 lastTick = IMBaseClient::gClient->getTime();
}

IMBaseOutEntity::IMBaseOutEntity(npsVec3f p,float search_r)
{
 roi = search_r;
 roi2 = roi*roi;
 primary_region.address = 0;
 address = 0;

 AvgBoundDia(1.0);
 AvgMaxVel(1.0);

 dead_timeout = 120.0;
 oldvel.makeNull();
 oldpos = pos = p;
 oldvel[0] = 0.0;  oldvel[1] = 0.0;  oldvel[2] = 0.0;
 obtain_address();
 lastTick = IMBaseClient::gClient->getTime();
 IMBaseClient::gClient->addOutEntity(this);
}
```

```
IMBaseOutEntity::~IMBaseOutEntity()
{
 IMBaseClient::gClient->remOutEntity(this);
 unsigned int old = primary_region.address;
 primary_region.address = 0;
 checkin(old);
}

void IMBaseOutEntity::obtain_address()
{
 IMBaseClient::gClient->MDHCP_getAddresses(1,&address);
 IMBaseClient::gClient->OCTREE_search_server(this);
}

void IMBaseOutEntity::checkin(unsigned int old)
{
 IMBaseClient::gClient->checkin(this,old,primary_region.address);
}

void IMBaseOutEntity::sendMoved(McastPacket &mpacket,unsigned int
send_to,unsigned int to,unsigned int from, short port)
{
 mpacket.type=McastPacket::MOVED;
 mpacket.address=to;
 mpacket.newaddress[0]=from;
 mpacket.newaddress[1]=address;
 mpacket.moved.x = pos[0];
 mpacket.moved.y = pos[1];
 mpacket.moved.z = pos[2];
 mpacket.moved.vx = vel[0];
 mpacket.moved.vy = vel[1];
 mpacket.moved.vz = vel[2];
 mpacket.moved.lastUpdate = lastUpdate;
 mpacket.moved.module_version = protocol_version();
 strcpy(mpacket.moved.module_name,protocol_name());
 strcpy(mpacket.moved.module_url,protocol_url());

 IMBaseClient::gClient->sendMoved(this,mpacket,send_to,to,from,port);
}

unsigned int IMBaseOutEntity::find_primary()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 set<IMNode *> new_nodes;

 static CellRegion cells[1024];
 static IMNode *nodes[1024];
 int num = IMBaseClient::gClient->tree->
           searchXYZR(pos[0],pos[1],pos[2],roi,cells,nodes);
 bool zr = false;
 for(int i=0;i<num;i++)
    {
     if (nodes[i])
        {
         if (nodes[i]->region.address != 0)
            new_nodes.insert(nodes[i]);
         else zr = true;
        }
    }
 if (zr)
    {
     printf("Found '0' Region. Must Search\n");
     IMBaseClient::gClient->OCTREE_search_server(this);
    }

 vector<IMNode *> add_nodes(new_nodes.size(),NULL);
 vector<IMNode *>::iterator a = add_nodes.begin();
 a = set_difference(new_nodes.begin(),new_nodes.end(),
                    current_nodes.begin(),current_nodes.end(),a);
 for(a = add_nodes.begin();a != add_nodes.end(); a++)
    if (*a)
       {
        (*a)->addtotree();
        current_nodes.insert(*a);
       }

 vector<IMNode *> old_nodes(current_nodes.size(),NULL);
 vector<IMNode *>::iterator o = old_nodes.begin();
 o = set_difference(current_nodes.begin(),current_nodes.end(),
                    new_nodes.begin(),new_nodes.end(),o);
 for(o = old_nodes.begin() ;o != old_nodes.end(); o++)
    if (*o)
       {
        (*o)->remfromtree();
        current_nodes.erase(*o);
       }

 return IMBaseClient::gClient->OCTREE_find_primary(this,true);
}

void IMBaseOutEntity::Update_dt()
{
 dt = IMBaseClient::gClient->getTime() - lastTick;
}

void IMBaseOutEntity::forceSend()
{
 double time = IMBaseClient::gClient->getTime();
 lastUpdate = time;
 oldvel = vel;
 oldpos = pos;
 static McastPacket p;
 this->sendMoved(p,primary_region.address,primary_region.address,0);
}

void IMBaseOutEntity::tick()
{
 double time = IMBaseClient::gClient->getTime();
 double big_dt = time-lastUpdate;

 int oldcell = find_primary();

 npsVec3f ghostpos = oldpos + oldvel * big_dt;
 vel = (pos - lastPos)/dt;

 if ((ghostpos - pos).lengthSqr() > dead_thresh ||
      big_dt > dead_timeout || oldcell)
    {
     lastUpdate = time;
     oldvel = vel;
     oldpos = pos;
     static McastPacket p;
     if (oldcell) checkin(oldcell);
     else this->sendMoved(p,primary_region.address,
                          primary_region.address,0);
    }

 lastPos = pos;
 lastTick = time;
}

void IMBaseOutEntity::calc_smallest_region()
{
 smallest_region = IMBaseClient::gClient->k1 * (avg_bound_dia +
                    avg_max_vel*IMBaseClient::gClient->MinTimeInRegion);

 printf("Smallest region now %f meters on a side.\n",smallest_region);
 dead_thresh =  IMBaseClient::gClient->k2 * smallest_region;

 printf("dead_thresh now %f meters.\n",dead_thresh);
 dead_thresh *= dead_thresh;
}
```

15. baseProtocol.c++

```
#include "IMBaseClient.h"
#include "IMBaseProtocol.h"
#include "IMBaseInEntity.h"

IMBaseInEntity *IMBaseProtocol::new_entity(McastPacket &mpacket)
{
 return new IMBaseInEntity(mpacket);
}

void IMBaseProtocol::del_entity(IMBaseInEntity *e)
{
 delete e;
}

void IMBaseProtocol::setup(void)
{
}
```

APPENDIX B. FISH ENTITY SOURCE CODE

A. INTRODUCTION

This appendix contains source-code for Bamboo module "IMFishModule" which contains the "fish" entities used within the experiment described in Chapter V, and the module "FishClient" which controlled 225 fish. For other examples on how to create and use protocol modules using the Three-Tiered interest management system see the NPSNET-V website (http://npsnet.org/~npsnet/v).

B. IMFISHMODULE

1. fishEntity.h

```
#ifndef _FISHENTITY
#define _FISHENTITY

#include "npsQuaternion.h"

class FishEntity
{
 protected:

   npsQuaternion orien;      // orientation

 public:

   static int rtt;
   static char module_name[64];
   static char module_url[192];
   static float module_version;

   npsVec3f dir;

   enum FishSize
   {
    FISH_SMALL,
    FISH_MEDIUM,
    FISH_LARGE,
    FISH_NOT_A_SIZE
   } size;
};
```

```
#endif
```

2. fishInEntity.h

```
#ifndef _FISHINENTITY
#define _FISHINENTITY

#include "IMPacket.h"
#include "fishEntity.h"
#include "IMBaseInEntity.h"

class FishInEntity : public IMBaseInEntity, public FishEntity
{
   npsQuaternion oldorien;
   npsQuaternion neworien;

 public:

   virtual int   runtime_type()      {return FishEntity::rtt;}
   virtual char *protocol_name()     {return FishEntity::module_name;}
   virtual char *protocol_url()      {return FishEntity::module_url;}
   virtual float protocol_version() {return FishEntity::module_version;}

   FishInEntity(McastPacket &mpacket) : IMBaseInEntity(mpacket)
                 { size=FISH_NOT_A_SIZE;update(mpacket); }

   virtual void update(McastPacket &mpacket);

   virtual void tick();

   virtual bool subscribe(IMBaseEntity *e);
   virtual bool unsubscribe(IMBaseEntity *e);
};

#endif
```

3. fishOutEntity.h

```
#ifndef _FISHOUTENTITY
#define _FISHOUTENTITY

#include <math.h>

#include "fishEntity.h"
#include "IMBaseOutEntity.h"

inline npsVec3f calcPos(float p,float r,npsVec3f c,npsVec3f n)
{
 npsVec3f np(r * sin(p) + c[0],c[1],r * cos(p) + c[2]);
 return np;
}

class FishOutEntity : public IMBaseOutEntity, public FishEntity
{
 private:

   npsVec3f norm; // Normal of plane
   npsVec3f cent; // Center of circle
   float radius;  // Radius of circle
   float phi;     // Current Angle (rads)

   // Per entity high quality dead reckoning
   double last_fish_Update;
   npsVec3f old_fish_vel;
   npsVec3f old_fish_pos;


 public:
   virtual bool interestedIn(IMBaseEntity *e);
   virtual void NotinterestedIn(IMBaseEntity *e);

   virtual int   runtime_type()     { return FishEntity::rtt; }
   virtual char *protocol_name()    { return FishEntity::module_name; }
   virtual char *protocol_url()     { return FishEntity::module_url; }
   virtual float protocol_version() { return FishEntity::module_version;
}

   virtual void sendMoved(McastPacket &mpacket,unsigned int send_to,
                          unsigned int to,unsigned int from,
                          short port = 9876);

   FishOutEntity(FishSize s,float search_r,float p,float r,
                 npsVec3f c,npsVec3f n);
   ~FishOutEntity();

   virtual void tick();// Update Entity
};

#endif
```

4. fishProtocol.h

```
#ifndef __FISHPROTOCOL__
#define __FISHPROTOCOL__

#include "bbModule.h"
#include "fishInEntity.h"
#include "IMBaseProtocol.h"
#include "IMSocket.h"

class bbThread;

class FishProtocol : public IMBaseProtocol
{
 private:
   bool cosmo;

   static void fish_loop(bbThread *,void *);

 public:
   Msocket *msocket;

   FishProtocol() : IMBaseProtocol("IMFishProtocolModule") {setup();}
   FishProtocol(const char *name) : IMBaseProtocol(name) {setup();}

   virtual void setup(void);

   virtual IMBaseInEntity *new_entity(McastPacket &packet)
   {
        return new FishInEntity(packet);
    }

    virtual void del_entity(IMBaseInEntity *e)
    {
        delete (FishInEntity *)e;
    }
};

#endif
```

5. fishInEntity.c++

```
#include "fishInEntity.h"
#include "fishProtocol.h"

void FishInEntity::update(McastPacket &mpacket)
{
 oldorien = neworien;
 npsVec3f forward(0.0,0.0,-1.0);
 neworien.makeFromVecs(forward, npsVec3f(mpacket.moved.vx,
                       mpacket.moved.v, mpacket.moved.vz));

 size = (FishSize)mpacket.moved.protocol_reserved[0];

 IMBaseInEntity::update(mpacket);
}

void FishInEntity::tick()
{
 IMBaseInEntity::tick();

 if (dt < smooth_time)
    {
     float frac = dt/smooth_time;
     orien.slerp(oldorien,neworien,frac,0);
    }
 else orien = neworien;
}

bool FishInEntity::subscribe(IMBaseEntity *e)
{
 static FishProtocol *protocol =
              (FishProtocol *)FishProtocol::findObject(protocol_name());

 if (!IMBaseInEntity::subscribe(e)) return false;
 protocol->msocket->subscribe(address);
 return true;
}

bool FishInEntity::unsubscribe(IMBaseEntity *e)
{
 static FishProtocol *protocol =
              (FishProtocol *)FishProtocol::findObject(protocol_name());

 if (!IMBaseInEntity::unsubscribe(e)) return false;
 if (subscribed()) return true;

 protocol->msocket->unsubscribe(address);
 return true;
}
```

6. fishOutEntity.c++

```
#include <math.h>

#include "IMBaseClient.h"
#include "fishOutEntity.h"
#include "fishInEntity.h"

FishOutEntity::FishOutEntity(FishSize s,float search_r,float p,
                             float r,npsVec3f c,npsVec3f n) :
               IMBaseOutEntity(calcPos(p,r,c,n),search_r,0.3,5,120.0)
{
 norm = n;
 cent = c;
 radius = r;
 phi = p;

 size = s;
 // Init direction
 if (dir[0] == 0 && dir[1] == 0 && dir[2] == 0)
    {
     dir[0] = 1.0;
     dir[1] = 1.0;
     dir[2] = 1.0;
     dir.normalize();
    }
}

bool FishOutEntity::interestedIn(IMBaseEntity *e)
{
 // We only like fish that are our size
 if (e->runtime_type() == FishEntity::rtt)
{
 if (e->mode() == INBOUND)
    {
     if (((FishInEntity *)e)->size == size)
        {
         if (!((FishInEntity *)e)->subscribed())
             ((FishInEntity *)e)->subscribe(this);
         return true;
        }
    }
 else
    {
     if (((FishOutEntity *)e)->size == size)
         return true;
    }
}

 if (e->mode() == INBOUND)
    if (((FishInEntity *)e)->subscribed())
       ((FishInEntity *)e)->unsubscribe(this);

 return false;
}

void FishOutEntity::NotinterestedIn(IMBaseEntity *e)
{
 if (e->runtime_type() == FishEntity::rtt)
    if (e->mode() == INBOUND)
       if (((FishInEntity *)e)->subscribed())
          ((FishInEntity *)e)->unsubscribe(this);
}

void FishOutEntity::tick()
{
 // Default speed...
 static float fish_vel = 5;

 IMBaseOutEntity::Update_dt();

 int num_inter = interestingEntities.size();
 if (num_inter)
    {
     map<IMBaseEntity *,float,IMBaseEntity::compare>::iterator i;

     //Calculated intersting things about intersting entities
     int numclose=0;
     npsVec3f pos_avg(0,0,0), dir_avg(0,0,0), avoid_avg(0,0,0);
     for(i=interestingEntities.begin();i!=interestingEntities.end();i++)
        {
         // Calculate group average position
         pos_avg = pos_avg + (*i).first->pos;

         // Calculate group average heading
         dir_avg = dir_avg + ((FishEntity *)(*i).first)->dir;

         // Calculate average avoidence heading
         float d2 = (*i).second;
         if (d2 < (20*20))
            {
             // Directly away
             npsVec3f dv = pos - (*i).first->pos;
             avoid_avg = avoid_avg + dv;
             numclose++;
            }
        }

     // Head toward Average position of interesting entities
     pos_avg = pos_avg / (float)num_inter;
     npsVec3f toAvg = pos_avg - pos;

     // Keep distance from other entities.
     if (numclose) avoid_avg = avoid_avg / (float)numclose;

     // Head in avg direction of entities
     dir_avg = dir_avg / (float)num_inter;

     // Merge weighted headings: This is our goal heading!
     npsVec3f goal = dir_avg * 0.35 + toAvg * 0.15 + avoid_avg * 0.5;
     goal.normalize();
     npsVec3f diff = (goal - dir);

     // At most 30deg / sec
     float len = diff.length();
     float max = (0.33 * dt);
     if (len > max) diff = diff * max / len;

     // Nudge our heading
     dir = dir + diff;
     dir.normalize();
    }

 // Update our position.
 pos = pos + dir * fish_vel * dt;

 // Keep them in 1km x 30 x 1km box
 if (pos[0] > 500) { pos[0] = 500; dir[0] = -dir[0];}
 if (pos[1] > 0) { pos[1] = 0; dir[1] = -dir[1];}
 if (pos[2] > 500) { pos[2] = 500; dir[2] = -dir[2];}
 if (pos[0] < -500) { pos[0] = -500; dir[0] = -dir[0];}
 if (pos[1] < -30) { pos[1] = -30; dir[1] = -dir[1];}
 if (pos[2] < -500) { pos[2] = -500; dir[2] = -dir[2];}

 //Do rest of update
 IMBaseOutEntity::tick();

 // Send out per entity mcast based on small thresholds
 double time = IMBaseClient::gClient->getTime();
 double big_dt = time-last_fish_Update;

 npsVec3f ghostpos = old_fish_pos + old_fish_vel * big_dt;
 if ((ghostpos - pos).lengthSqr() > 1.0 || big_dt > 5.0)
    {
     last_fish_Update = time;
     old_fish_vel = vel;
     old_fish_pos = pos;
     static McastPacket p;
     sendMoved(p,address,address,0,9090);
    }
}

void FishOutEntity::sendMoved(McastPacket &mpacket,unsigned int
send_to,unsigned int to,unsigned int from,short port)
{
 mpacket.moved.protocol_reserved[0] = (char)size;
 IMBaseOutEntity::sendMoved(mpacket,send_to,to,from,port);
}

FishOutEntity::~FishOutEntity()
{
 IMBaseClient::gClient->remOutEntity(this);
 unsigned int old = primary_region.address;
 primary_region.address = 0;
 checkin(old);
}
```

7. fishProtocol.c++

```
#include "IMBaseClient.h"
#include "fishEntity.h"
#include "fishProtocol.h"

#include "statBaseClient.h"
#include "stat_recorder.h"

#include "bbThread.h"

int FishEntity::rtt;
```

```
char FishEntity::module_name[64] = "IMFishProtocolModule";
char FishEntity::module_url[192] = "";
float FishEntity::module_version = 0.894;

extern "C"
{
 static FishProtocol *proto = NULL;
 static bbThread *loop = NULL;

 bool initFunc(void)
 {
  proto = new FishProtocol();
  FishEntity::rtt = IMBaseClient::getNextRTT();
  return true;
 }

 bool exitFunc(void)
 {
  delete proto;
  delete loop;
  return true;
 }
}

void FishProtocol::fish_loop(bbThread *,void *)
{
 static McastPacket mpacket;
 static IMBaseInEntity *ng = new IMBaseInEntity(mpacket);
 static set<IMBaseInEntity *,IMBaseEntity::compare>::iterator i;
 static stat_recorder *pps = bbModule::findObject("IMStatClientModule")?
            (((IMStatClient*)IMBaseClient::gClient)->pps_recorder):NULL;

 ACE_OS::recvfrom(proto->msocket->get_handle(),(char *)&mpacket,
                  sizeof(McastPacket), 0,0,0);
 if (pps) pps->record_count();
 ng->address = mpacket.newaddress[1];

 if ((i = IMBaseClient::gClient->inlist.find(ng)) ==
      IMBaseClient::gClient->inlist.end())
     return;

 // Update his packet unless we have a newer update
 if ((*i)->lastUpdate < mpacket.moved.lastUpdate)
     ((FishInEntity *)(*i))->update(mpacket);
}

void FishProtocol::setup(void)
{
 cosmo = bbModule::findObject("npsCosmo3dModule")?true:false;

 msocket = new Msocket(9090);

 //Spawn fish network thread.
 loop = new bbThread(fish_loop);
}
```

C. FISHCLIENT

1. client.c++

```
#include<stdio.h>
#include<stdlib.h>

#include "bbThread.h"
#include "bbCommandLine.h"

#include "IMBaseClient.h"
#include "fishOutEntity.h"

#define NUM_ENTITY 225

bbThread *tickThread;
FishOutEntity *ent[NUM_ENTITY];

extern "C"
{
void tickFunc(bbThread *thread, void *data)
{
 IMBaseClient::gClient->tick();
}

void runme()
{
 //We decide what type of Client the global one is!
 IMBaseClient::gClient = new IMBaseClient();

 const char *myname = bbCommandLine::getModule(1);

 npsVec3f c((float)atoi(bbCommandLine::getModuleArg(myname,0)+1),
            (float)atoi(bbCommandLine::getModuleArg(myname,1)+1),
            (float)atoi(bbCommandLine::getModuleArg(myname,2)+1));

 npsVec3f n(0.0,1.0,0.0);
 float r,p;
 for(int i=0;i<NUM_ENTITY;i+=3)
    {
     r = rand()%100;
     p = (rand()%628)/100.0;
     ent[i] = new FishOutEntity(FishOutEntity::FISH_SMALL,35.0,p,r,c,n);

     r = rand()%100;
     p = (rand()%628)/100.0;
    ent[i+1]=new FishOutEntity(FishOutEntity::FISH_MEDIUM,35.0,p,r,c,n);

     r = rand()%100;
     p = (rand()%628)/100.0;
     ent[i+2]=new FishOutEntity(FishOutEntity::FISH_LARGE,35.0,p,r,c,n);
    }

 tickThread = new bbThread(tickFunc,NULL);
}
void killme()
{
 printf("Unloading fish test module.\n");
 delete tickThread;
 printf("Killed tick thread\n");
 for(int c=0;c<NUM_ENTITY;c++)
    {
     delete ent[c];
    }
 printf("deleted Entities\n");
 delete IMBaseClient::gClient;
 printf("Killed gClient\n");
 printf("Unloaded boid test module.\n");
}
      }
```

APPENDIX C. BROADCAST SYSTEM SOURCE CODE

A. INTRODUCTION

This appendix contains source code for the broadcast-based filtering client used in the experiment described in Chapter V. This implementation was only used for comparisons against the Three-Tier system, and is only provided here for completeness.

B. BROADCAST CLIENT

1. IMBaseClient.h

```
#ifndef _IMBASECLIENT
#define _IMBASECLIENT
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <net/if.h>
#include <net/soioctl.h>
#include <arpa/inet.h>

#include <multimap.h>
#include <set.h>
#include "bbMutex.h"
#include "IMBaseEntity.h"
#include "IMBaseOutEntity.h"

//Forward declare classes instead of including them
//This saves LOTS of memory when using Bamboo modules
class bbThread;
class Msocket;
class IMBaseInEntity;
class IMBaseProtocol;
class IMNode;

class IMBaseClient
{
 public:
   static  void readFunc(bbThread *, bbData *);
   virtual void mcastcheck();

   bbMutex inlistLock;
 public:
  int bigread(int fd,void *buf,int num)
  {
   int cnt = 0;
   while(cnt < num && cnt != -1)
         cnt += read(fd,(char *)buf+cnt,num-cnt);
   return cnt;
  }

  int bigwrite(int fd,void *buf,int num)
  {
   int cnt = 0;
   while(cnt < num && cnt != -1)
         cnt += write(fd,(char *)buf+cnt,num-cnt);
   return cnt;
  }

  int s2;            //mcast socket for sending

  IMNode *tree;

  IMBaseProtocol *base_proto;

  void setup();

  bool auto_tick;      // Should we automaticly tick outentities?
  bool internal_clock; // Should we set our time based on gettimeofday?
  double simtime;      // Simulation Time

  set<IMBaseOutEntity *,IMBaseEntity::compare> outlist;
  set<IMBaseInEntity *,IMBaseEntity::compare> inlist;

  bbThread *readThread;

  virtual void filter(McastPacket &mpacket,IMBaseInEntity *ng);

  typedef pair<unsigned int,McastPacket> packet_type;
  multimap<double,packet_type> delaySend;
  multimap<double,unsigned int> delaySplit,delayMerge;
  multimap<double,IMBaseOutEntity *> delayRespond;

  unsigned int clientaddress;

public:

  Msocket *msocket;
  static IMBaseClient *gClient;

  IMBaseClient();
  ~IMBaseClient();

  virtual void tick();
  double getTime() { return simtime; }
  void setTime(double t) { simtime = t; }

  static int numrtt;
  static int getNextRTT() { numrtt++; return numrtt-1; }

  virtual void addOutEntity(IMBaseOutEntity *e) { outlist.insert(e); }
  virtual void remOutEntity(IMBaseOutEntity *e) { outlist.erase(e); }
```

```
   virtual void checkin(IMBaseOutEntity *e,unsigned int from,
                        unsigned int to);
   virtual void sendMoved(IMBaseOutEntity *e,McastPacket &mpacket,
                          unsigned int send_to,unsigned int to,
                          unsigned int from);
   virtual void sendPing(unsigned int send_to);
   virtual void sendMcast(unsigned int send_to,McastPacket &packet);
   virtual void sendDelayedMcast(unsigned int send_to,
                                 McastPacket &packet,double delay);

   virtual unsigned int OCTREE_search_server(IMBaseOutEntity *e);
   virtual unsigned int OCTREE_find_primary(IMBaseOutEntity *e,
                                            bool ask);
   virtual unsigned int OCTREE_ask_friends(IMBaseOutEntity *e,
                                           unsigned int address);
   virtual void OCTREE_Merge(IMNode *n);
   virtual void OCTREE_Split(IMNode *n);

   virtual bool MDHCP_getAddresses(int num,unsigned int *addrs);
};

#endif
```

2. IMBaseEntity.h

```
#ifndef _IMBASEENTITY
#define _IMBASEENTITY

#include <set.h>

#include "IMPacket.h"

#include "npsVec3f.h"
#include "IMCellRegion.h"

class IMBaseEntity
{
 public:

   unsigned int address; // Server assigned multicast address or guid.

   npsVec3f pos;         // position
   npsVec3f vel;         // velocity

   double lastUpdate;    // Time of last update (secs past Jan 1, 1970)

   CellRegion primary_region; //copy of primary region

 protected:

   npsVec3f oldpos;      // Position at time = lastUpdate

 public:

   enum { INBOUND,OUTBOUND };

   virtual int   mode() = 0;

   virtual int   runtime_type()     { return rtt; }
   virtual char *protocol_name()    { return module_name; }
   virtual char *protocol_url()     { return module_url; }
   virtual float protocol_version() { return module_version; }

   static int rtt;
   static char module_name[64];
   static char module_url[192];
   static float module_version;

   virtual void tick() = 0; // Update Entity

   virtual bool interestedIn(IMBaseEntity *e) { return false; }

   struct compare :
   public binary_function<IMBaseEntity *,IMBaseEntity *,bool>
   {
    bool operator()(IMBaseEntity *r1,IMBaseEntity *r2) const
              {
               return (r1->address < r2->address);
              }
   };
};

#endif
```

3. IMBaseInEntity.h

```
#ifndef _IMBASEINENTITY
#define _IMBASEINENTITY

#include "IMBaseEntity.h"
#include "IMPacket.h"

#include "string"
#include "map.h"

class IMBaseInEntity : public IMBaseEntity
{
 protected:

   int internal;  // ref count; 0 == entity updates come from octree

   double lastlastUpdate;
   npsVec3f oldoldpos;
   npsVec3f oldvel;
   float smooth_time;

   double dt;

   McastPacket lastPacket;

 public:

   IMBaseInEntity(McastPacket &packet)
   {
    internal = 0;
    address = packet.newaddress[1];
    update(packet);
   }

   virtual int mode() { return INBOUND; }
   virtual void tick(); // Update Entity
   virtual void update(McastPacket &mpacket);

   virtual bool subscribed() { return (internal > 0); }
   virtual void subscribe() { internal++; }
   virtual void unsubscribe() { internal--; }

   map<string,void *> attributeMap;
};

#endif
```

4. IMBaseOutEntity.h

```
#ifndef _IMBASEOUTENTITY
#define _IMBASEOUTENTITY

#include "npsVec3f.h"
#include "IMBaseEntity.h"
#include "IMBaseProtocol.h"
#include "IMCellRegion.h"

#include "map.h"

class IMBaseOutEntity : public IMBaseEntity
{
 protected:

   float dead_thresh;           // Update if error > dead_thresh (m^2)
   float dead_timeout;          // Update if time > dead_timeout (s)

   void obtain_address();
   void checkin(unsigned int old);
   virtual void sendMoved(McastPacket &mpacket,unsigned int send_to,
                          unsigned int to,unsigned int from);

   unsigned int find_primary();

   npsVec3f lastPos;

 public:

   ///////////////////////////////////////////////
   // IE Parameters
   //
   bool SubUnknownProtocols; // Should we by defualt subscribe to
                             // entities we did not have an IE for?

   float smallest_entity;    // Smallest entity we should look for.
                             // (Similar to smallest region calc)
   float smallest_entity_avg_bound_dia; // Used in smallest_entity calc.
   float smallest_entity_avg_max_vel;   // Used in smallest_entity calc.

   float roi;   //Tier 2 - Radius of interest
   float roi2;  //Tier 2 - Radius of interest squared


   map<IMBaseEntity *,float,IMBaseEntity::compare> interestingEntities;
```

```
  protected:

    void calc_smallest_region();
    double lastTick;                  // Time this guy was ticked last
    double dt;                        // currenttime - lastTick

    void Update_dt();                 // Updates dt;

    CellRegion region;  // Current location
    npsVec3f oldvel;    // Velocity at time = lastUpdate

  public:

    virtual int mode() { return OUTBOUND; }

    virtual void forceSend(); // Used for pings
    virtual void tick();      // Update Entity

    IMBaseOutEntity(npsVec3f p, float search_r, float dia,
                    float vel,float timeout);
    IMBaseOutEntity(npsVec3f p, float search_r);

    ~IMBaseOutEntity();
};
#endif
```

5. IMBaseNode.h

```
#ifndef _IMBASENODE
#define _IMBASENODE

#include "set.h"

#include "IMCellRegion.h"

//Forward declare classes
class IMBaseOutEntity;

class IMNode
{
 public:
   CellRegion region; //Bounds
   IMNode *parent;
   IMNode *child[8];
   bool leaf;          //True if node has no children

   set<unsigned int> elist;

   IMNode(CellRegion r)
   {
    parent=NULL;
    leaf=true;
    for(int i=0;i<8;i++) child[i]=NULL;
    memcpy(&region,&r,sizeof(CellRegion));
   }

   IMNode()
   {
    parent=NULL;
    leaf=true;
    for(int i=0;i<8;i++) child[i]=NULL;
   }

   //Number on entities in child and parent
   //Returns -1 if all children present && not leaf nodes!
   virtual int childrenSize();

   virtual void splittree(unsigned int newaddr[8]);
   virtual void mergetree();
   virtual void mergeregions(CellRegion *region);
   virtual void fleshtree();
   virtual IMNode *findnode(unsigned int address);
   virtual IMNode *addtotree(CellRegion r);
   virtual IMNode *XYZtonode(IMBaseOutEntity *e);
   virtual int searchXYZR(double x,double y,double z,double r,
                          CellRegion *cells);

};

#endif
```

6. IMBaseProtocol.h

```
#ifndef __IMBASEPROTOCOL__
#define __IMBASEPROTOCOL__

#include "bbMappedClass.h"

#include "IMPacket.h"

// forward declare
class IMBaseInEntity;

class IMBaseProtocol : public bbMappedClass<IMBaseProtocol>
{
 public:

   IMBaseProtocol(): bbMappedClass<IMBaseProtocol>("IMBaseClientModule")
                   {setup();}
   IMBaseProtocol(const char *name): bbMappedClass<IMBaseProtocol>(name)
                   {setup();}

   virtual void setup(void);

   virtual IMBaseInEntity *new_entity(McastPacket &mpacket);
   virtual void del_entity(IMBaseInEntity *);
};

class IMBaseProtocolIE
{
 public:
   IMBaseProtocol *protocol; //Which protocol is this for

   struct compare :
   public binary_function<IMBaseProtocolIE *,IMBaseProtocolIE *,bool>
   {
    bool operator()(IMBaseProtocolIE *r1,IMBaseProtocolIE *r2) const
                {
                 return (r1->protocol < r2->protocol);
                }
   };
};
#endif
```

7. IMCellRegion.h

```
#ifndef __IM_CELLREGION
#define __IM_CELLREGION

#include "npsVec3f.h"
#include "IMExtent.h"

struct CellRegion
{
  unsigned int address;
  Extent ext;
  int below;

  inline bool contains(npsVec3f p)
  {
   if (p[0] <= ext.x + ext.r && p[0] > ext.x - ext.r &&
       p[1] <= ext.y + ext.r && p[1] > ext.y - ext.r &&
       p[2] <= ext.z + ext.r && p[2] > ext.z - ext.r) return true;
   return false;
  }
};

#endif
```

8. IMExtent.h

```
#ifndef __IM_EXTENT
#define __IM_EXTENT

// Messages from server
struct Extent
{
 double x;
 double y;
 double z;
 double r;
};

#endif
```

9. IMPacket.h

```
#ifndef __IM_PACKET
#define __IM_PACKET

#include "IMCellRegion.h"

// Messages to server
struct ServerPacket
{
 enum PacketType {SEARCH,CHANGE,MALLOC,LOCK,SPLIT,MERGE};
 enum PacketType type;
 union
 {
  struct
  {
   int num;
  } malloc;

  struct
  {
   unsigned int address;
   unsigned int newaddr[8];
  } split;

  struct
  {
   unsigned int address;
   double x;
   double y;
   double z;
   double r;
  } search;

  struct
  {
   unsigned int from;
   unsigned int to;
  } change;

 };
};

// Messages to mcast address
struct McastPacket
{
 enum PacketType {MOVED,PINGING,MERGE,SPLIT,QUERY,RESPOND};
 enum PacketType type;
 unsigned int address;
 union
     {
      unsigned int newaddress[8]; //XXXX is this used?!
      struct
           {
            unsigned int newaddress[2];
            double lastUpdate;
            double x;
            double y;
            double z;
            float vx;
            float vy;
            float vz;
            char module_name[64];
            char module_url[192];
            float module_version;
            char protocol_reserved[712];
           } moved; // 312 + 712 = 1024

      struct
           {
            unsigned int tcpaddress;
            unsigned int port;
            double r;
            double x;
            double y;
            double z;
           } query; // 40 bytes
      };
};

#endif
```

10. IMSocket.h

```
#ifndef _MSOCKET
#define _MSOCKET

#include <set.h>

#include "ace/SOCK_Dgram.h"

#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */

#include "ace/INET_Addr.h"

//A Socket that reads from multiple mcast addresses

class ACE_Export Msocket : public ACE_SOCK_Dgram
{
 public:
   Msocket(unsigned short port);
   ~Msocket(void);

   bool subscribe(unsigned int address);
   void unsubscribe(unsigned int address);

 private:
   set<unsigned int> sub_list;
   in_addr ifaddr;
};

#endif
```

11. baseClient.c++

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#include <sys/time.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "bbPrinter.h"
#include "bbThread.h"

#include "IMBaseClient.h"
#include "IMBaseOutEntity.h"
#include "IMBaseInEntity.h"
```

```
#include "IMBaseProtocol.h"

#include "IMSocket.h"
#include "ace/Handle_Set.h"

#include "IMPacket.h"
#include "IMBaseNode.h"

#define JOIN_DELAY 10

int IMBaseClient::numrtt = 0;
IMBaseClient *IMBaseClient::gClient;

IMBaseClient::IMBaseClient()
{
 internal_clock = true;
 auto_tick = true;

 struct timeval tt;
 gettimeofday(&tt);
 setTime((double)tt.tv_sec + tt.tv_usec/1000000.0);

 outlist.clear();
 inlist.clear();

 // Set up the multicast socket
 int on=1,ttl = 15;
 unsigned char off = 0;
 msocket = new Msocket(9976);
 msocket->subscribe(inet_addr("239.0.0.1"));
 s2 = socket(AF_INET, SOCK_DGRAM, 0);
 setsockopt(s2, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
 if (setsockopt(s2, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on)) < 0)
    perror("setsockopt SO_REUSEPORT");
 if (setsockopt(s2, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
    perror("setsockopt SO_REUSEADDR");
 if (setsockopt(s2, IPPROTO_IP, IP_MULTICAST_LOOP, &off, sizeof(off))<0)
    perror("setsockopt IP_MULTICAST_LOOP");

 //Turn on the base protocol. This is our default entity.
 base_proto = new IMBaseProtocol();
 IMBaseEntity::rtt = IMBaseClient::getNextRTT();

 readThread = new bbThread(readFunc,NULL);
}


IMBaseClient::~IMBaseClient()
{
 printf("IMBaseClient shutting down\n");
 delete readThread;
 printf("Killed readThread.\n");
 delete msocket;
 printf("deleted msocket.\n");
 printf("IMBaseClient shutdown\n");
}

void IMBaseClient::tick()
{
 struct timeval tt;
 gettimeofday(&tt);
```

```
  double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;

  // Go through delay send queue
  multimap<double,packet_type>::iterator p = delaySend.begin();
  while(p != delaySend.end() && (*p).first < tod)
       {
        sendMcast((*p).second.first,(*p).second.second);
        delaySend.erase(p);
        p = delaySend.begin();
       }

  //Set simtime based on tod
  if (internal_clock) setTime(tod);

  set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i;
  set<IMBaseInEntity *,IMBaseEntity::compare>::iterator j;

  // This manages the interestingEntities list for inbounds.
  for(j = inlist.begin();j!=inlist.end();j++)
     for(i = outlist.begin();i!=outlist.end();i++)
        {
         float dist2 = ((*i)->pos - (*j)->pos).lengthSqr();
         if (dist2 < (*i)->roi2 && (*i)->interestedIn(*j))
            (*i)->interestingEntities.insert(
                  pair<IMBaseEntity *,float>(*j,dist2));
         else
            (*i)->interestingEntities.erase(*j);
        }

  // This manages the interestingEntities list for outbounds.
  for(i = outlist.begin();i!=outlist.end();i++)
     {
      set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i2 = i;
      for(i2++;i2!=outlist.end();i2++)
         {
          float dist2 = ((*i)->pos - (*i2)->pos).lengthSqr();
          if (dist2 < (*i)->roi2 && (*i)->interestedIn(*i2))
             (*i)->interestingEntities.insert(
                   pair<IMBaseEntity *,float>(*i2,dist2));
          else
             (*i)->interestingEntities.erase(*i2);
          if (dist2 < (*i2)->roi2 && (*i2)->interestedIn(*i))
             (*i2)->interestingEntities.insert(
                    pair<IMBaseEntity *,float>(*i,dist2));
          else
             (*i2)->interestingEntities.erase(*i);
         }
     }

  if (auto_tick)
     {
      for(i = outlist.begin();i!=outlist.end();i++)
         (*i)->tick();

      for(j = inlist.begin();j!=inlist.end();j++)
         (*j)->tick();
     }
}
```

```
void IMBaseClient::sendDelayedMcast(unsigned int send_to,McastPacket
&packet,double delay)
{
 struct timeval tt;
 gettimeofday(&tt);
 double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
 packet_type packetPair(send_to,packet);
 multimap<double,packet_type>::
              value_type valuePair(delay+tod,packetPair);
 delaySend.insert(valuePair);
}

void IMBaseClient::sendMcast(unsigned int send_to,McastPacket &packet)
{
 struct sockaddr_in sa_mcast;
 sa_mcast.sin_family=AF_INET;
 struct in_addr grpaddr;
 grpaddr.s_addr = send_to;
 sa_mcast.sin_addr=grpaddr;
 sa_mcast.sin_port=htons(9976);

 sendto(s2,&packet,sizeof(McastPacket),0,&sa_mcast,sizeof(sa_mcast));
}

void IMBaseClient::sendMoved(IMBaseOutEntity *e,McastPacket
&mpacket,unsigned int send_to,unsigned int to,unsigned int from)
{
 sendMcast(send_to,mpacket);
}

void IMBaseClient::filter(McastPacket &mpacket,IMBaseInEntity *ng)
{
 //Does Protocol Exist? O(lg P)
 IMBaseProtocol *protocol =
IMBaseProtocol::findObject(mpacket.moved.module_name);
 if (protocol == NULL)
    {
     bbModule::load(mpacket.moved.module_name,
               mpacket.moved.module_version,0,mpacket.moved.module_url);
     protocol = IMBaseProtocol::findObject(mpacket.moved.module_name);

     if (protocol == NULL)
        {
         protocol = base_proto;
        }
    }

 //Does Entity Exist?
 set<IMBaseInEntity *,IMBaseEntity::compare>::iterator i;

 inlistLock.acquire();
 // Yes - an old guy. O(lg N)
 if ((i = inlist.find(ng)) != inlist.end())
    {
     if ((*i)->lastUpdate < mpacket.moved.lastUpdate)
        (*i)->update(mpacket);
    }
 else // No - a new guy O(lg N)
    {
     i = inlist.insert(protocol->new_entity(mpacket)).first;
    }
 // If he is checking out, delete him!
 if (mpacket.address == 0)
    {
     printf("Killing entity...\n");
     IMBaseInEntity *tmp = *i;
     inlist.erase(i);
     protocol->del_entity(tmp);
    }
 inlistLock.release();
}

void IMBaseClient::readFunc(bbThread *,bbData *that)
{
 if (IMBaseClient::gClient != NULL)
    IMBaseClient::gClient->mcastcheck();
 else
    sleep(0);
}

void IMBaseClient::mcastcheck()
{
 int rval;
 McastPacket mpacket;
 ACE_Handle_Set handle_set;

 handle_set.reset();
 handle_set.set_bit(msocket->get_handle());

 // Time out so thread will unblock.
 ACE_Time_Value t2(1,0);

 if ((rval = ACE_OS::select((msocket->get_handle())+1,handle_set,
      NULL,NULL,&t2)) != 0)
    {
     if (rval == -1) perror("select");
     else
        {
         ACE_OS::recvfrom(msocket->get_handle(),(char *)&mpacket,
                          sizeof(McastPacket),0,0,0);
         switch (mpacket.type)
                {
                 case McastPacket::MOVED:
                          {
                           static IMBaseInEntity *ng =
                                    new IMBaseInEntity(mpacket);
                           ng->address = mpacket.newaddress[1];
                           filter(mpacket,ng);
                          }
                          break;

                 default: printf("Unknown message from socket!\n");
                }
           }
    }
}
```

12. baseInEntity.c++

```
#include "IMBaseEntity.h"

int IMBaseEntity::rtt;
char IMBaseEntity::module_name[64] = "IMBaseClientModule";
char IMBaseEntity::module_url[192] = "";
float IMBaseEntity::module_version = 0.894;

#include "IMBaseClient.h"
#include "IMBaseInEntity.h"

void IMBaseInEntity::update(McastPacket &mpacket)
{
 memcpy(&lastPacket,&mpacket,sizeof(McastPacket));
 npsVec3f newpos(mpacket.moved.x,mpacket.moved.y,mpacket.moved.z);
 npsVec3f newvel(mpacket.moved.vx,mpacket.moved.vy,mpacket.moved.vz);

 oldoldpos = oldpos;
 oldpos = newpos;

 oldvel = vel;
 vel = newvel;

 lastlastUpdate = lastUpdate;
 lastUpdate = mpacket.moved.lastUpdate;
}

void IMBaseInEntity::tick()
{
 dt = IMBaseClient::gClient->getTime()-lastUpdate;
 pos = oldpos + vel * dt;

 if (dt < smooth_time)
     {
      double dt2 = IMBaseClient::gClient->getTime()-lastlastUpdate;
      float frac = dt/smooth_time;
      pos = pos * frac + (oldoldpos + oldvel * dt2) * (1.0f-frac);
     }
}
```

13. baseOutEntity.c++

```
#include "IMBaseClient.h"
#include "IMBaseOutEntity.h"

IMBaseOutEntity::IMBaseOutEntity(npsVec3f p, float search_r, float dia,
                                  float vel, float timeout)
{
 roi = search_r;
 roi2 = roi*roi;
 primary_region.address = 0;
 address = 0;

 AvgBoundDia(dia);
 AvgMaxVel(vel);

 dead_timeout = 5.0;
 oldvel.makeNull();
 oldpos = pos = p;
 oldvel[0] = 0.0;  oldvel[1] = 0.0;  oldvel[2] = 0.0;
 obtain_address();
 IMBaseClient::gClient->addOutEntity(this);
 lastTick = IMBaseClient::gClient->getTime();
}

IMBaseOutEntity::IMBaseOutEntity(npsVec3f p,float search_r)
{
 roi = search_r;
 roi2 = roi*roi;
 primary_region.address = 0;
 address = 0;

 AvgBoundDia(1.0);
 AvgMaxVel(1.0);

 dead_timeout = 5.0;
 oldvel.makeNull();
 oldpos = pos = p;
 oldvel[0] = 0.0;  oldvel[1] = 0.0;  oldvel[2] = 0.0;
 obtain_address();
 lastTick = IMBaseClient::gClient->getTime();
 IMBaseClient::gClient->addOutEntity(this);
}

IMBaseOutEntity::~IMBaseOutEntity()
{
 IMBaseClient::gClient->remOutEntity(this);
 unsigned int old = primary_region.address;
 primary_region.address = 0;
 checkin(old);
}

void IMBaseOutEntity::obtain_address()
{
 static unsigned int clientaddress = 0;
 if (!clientaddress)
    {
     char myname[255];
     gethostname(myname,255);
     struct hostent *en = gethostbyname(myname);
     clientaddress = *((int *)(en->h_addr_list[0]));
     clientaddress = (clientaddress >> 24) + (clientaddress<<24) +
                     (clientaddress&0x00ffff00);
    }

 address = clientaddress++;
}

void IMBaseOutEntity::checkin(unsigned int old)
{
}

void IMBaseOutEntity::sendMoved(McastPacket &mpacket,unsigned int
send_to,unsigned int to,unsigned int from)
{
 mpacket.type=McastPacket::MOVED;
 mpacket.address=to;
 mpacket.newaddress[0]=from;
 mpacket.newaddress[1]=address;
 mpacket.moved.x = pos[0];
 mpacket.moved.y = pos[1];
 mpacket.moved.z = pos[2];
 mpacket.moved.vx = vel[0];
 mpacket.moved.vy = vel[1];
 mpacket.moved.vz = vel[2];
 mpacket.moved.lastUpdate = lastUpdate;
 mpacket.moved.module_version = protocol_version();
 strcpy(mpacket.moved.module_name,protocol_name());
 strcpy(mpacket.moved.module_url,protocol_url());

 IMBaseClient::gClient->sendMoved(this,mpacket,send_to,to,from);
}

unsigned int IMBaseOutEntity::find_primary()
{
 return 0;
}

void IMBaseOutEntity::Update_dt()
{
 dt = IMBaseClient::gClient->getTime() - lastTick;
}

void IMBaseOutEntity::forceSend()
{
 double time = IMBaseClient::gClient->getTime();
 lastUpdate = time;
 oldvel = vel;
 oldpos = pos;
 sendMoved(McastPacket(),primary_region.address,
          primary_region.address,0);
}

void IMBaseOutEntity::tick()
{
 double time = IMBaseClient::gClient->getTime();
 double big_dt = time-lastUpdate;

 // DIS doesn't change primary_regions.
 primary_region.address = inet_addr("239.0.0.1");

 npsVec3f ghostpos = oldpos + oldvel * big_dt;
 vel = (pos - lastPos)/dt;

 if ((ghostpos - pos).lengthSqr() > dead_thresh || big_dt >
dead_timeout)
    {
     lastUpdate = time;
     oldvel = vel;
     oldpos = pos;
     sendMoved(McastPacket(),primary_region.address,
              primary_region.address,0);
    }

 lastPos = pos;
 lastTick = time;
}

void IMBaseOutEntity::calc_smallest_region()
{
 dead_thresh = 1.0;
 printf("dead_thresh now %f meters.\n",dead_thresh);
}
```

14. baseProtocol.c++

```
#include "IMBaseClient.h"
#include "IMBaseProtocol.h"
#include "IMBaseInEntity.h"

IMBaseInEntity *IMBaseProtocol::new_entity(McastPacket &mpacket)
{
 return new IMBaseInEntity(mpacket);
}

void IMBaseProtocol::del_entity(IMBaseInEntity *e)
{
 delete e;
}

void IMBaseProtocol::setup(void)
{
}
```

15. msocket.c++

```
#include "IMSocket.h"

Msocket::Msocket(unsigned short port)
{
if (ACE_SOCK_Dgram::open(ACE_INET_Addr(port),PF_INET,0,1) < 0)
    perror("open:");

 char buf[BUFSIZ];
 struct ifconf ifc;
 ifc.ifc_len = sizeof(buf);
 ifc.ifc_buf = buf;
 if (ACE_OS::ioctl(get_handle(), SIOCGIFCONF, &ifc) < 0)
    perror("ioctl SIOCGIFCONF");
 struct ifreq *ifr;
 ifr = ifc.ifc_req;
 ifaddr = ((struct sockaddr_in *)&ifr->ifr_addr)->sin_addr;

 int on = 1;
 if (set_option(SOL_SOCKET,SO_REUSEADDR,&on,sizeof on) == -1)
    perror("setsockopt SO_REUSEADDR");
 if (set_option(SOL_SOCKET,SO_REUSEPORT,&on,sizeof on) == -1)
    perror("setsockopt SO_REUSEPORT");

 int off = 0;
 if (set_option(IPPROTO_IP, IP_MULTICAST_LOOP, &off, sizeof off) == -1)
    perror("msocket setsockopt IP_MULTICAST_LOOP");
}

Msocket::~Msocket()
{
 ACE_SOCK_Dgram::close();
}
```

```
bool Msocket::subscribe(unsigned int address)
{
 //Don't bother if already subscribed
 if (!sub_list.insert(address).second) return false;

 ip_mreq mreq;
 mreq.imr_multiaddr.s_addr = address;
 mreq.imr_interface = ifaddr;
 if (ACE_SOCK::set_option(IPPROTO_IP,IP_ADD_MEMBERSHIP,
     &mreq,sizeof mreq)== -1)
    perror("setsockopt add membership");

 return true;
}

void Msocket::unsubscribe(unsigned int address)
{
 //Don't bother if not already subscribed
 if (!sub_list.erase(address)) return;

 in_addr grpaddr;
 ip_mreq mreq;
 grpaddr.s_addr = address;
 mreq.imr_multiaddr = grpaddr;
 mreq.imr_interface = ifaddr;
 if (ACE_SOCK::set_option(IPPROTO_IP,IP_DROP_MEMBERSHIP,
     &mreq,sizeof mreq)== -1)
    perror("setsockopt add membership");
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. REGION SYSTEM SOURCE CODE

A. INTRODUCTION

This appendix contains source code for the region-based filtering client used in the experiment described in Chapter V. This implementation was only used for comparisons against the Three-Tier system, and is only provided here for completeness.

B. REGIONMODULE

1. IMBaseClient.h

```
#ifndef _IMBASECLIENT
#define _IMBASECLIENT
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <net/if.h>
#include <net/soioctl.h>
#include <arpa/inet.h>

#include <multimap.h>
#include <set.h>
#include "bbMutex.h"
#include "IMBaseEntity.h"
#include "IMBaseOutEntity.h"

//Forward declare classes instead of including them
//This saves LOTS of memory when using Bamboo modules
class bbThread;
class Msocket;
class IMBaseInEntity;
class IMBaseProtocol;
class IMNode;

#define REGION_SIZE 4475
#define WORLDSIZE 4

class IMBaseClient
{
 public:
   static  void readFunc(bbThread *, bbData *);
   virtual void mcastcheck();
```

```
  bbMutex tree_mutex;

public:

  int bigread(int fd,void *buf,int num)
  {
   int cnt = 0;
   while(cnt < num && cnt != -1)
         cnt += read(fd,(char *)buf+cnt,num-cnt);
   return cnt;
  }

  int bigwrite(int fd,void *buf,int num)
  {
   int cnt = 0;
   while(cnt < num && cnt != -1)
         cnt += write(fd,(char *)buf+cnt,num-cnt);
   return cnt;
  }

  int s2;           //mcast socket for sending

  IMNode *tree[WORLDSIZE*2][WORLDSIZE*2][WORLDSIZE*2];

  IMBaseProtocol *base_proto;

  struct sockaddr_in sa;    // sockaddr_in used to connect to server
  char server_name[255];
  short server_port;

  virtual void setup();

  bool auto_tick;      // Should we automaticly tick outentities?
  bool internal_clock; // Should we set our time based on gettimeofday?
  double simtime;      // Simulation Time

  set<IMBaseOutEntity *,IMBaseEntity::compare> outlist;
  set<IMBaseInEntity *,IMBaseEntity::compare> inlist;
  bbThread *readThread;

  virtual void filter(McastPacket &mpacket,IMBaseInEntity *ng);

  typedef pair<unsigned int,McastPacket> packet_type;
  multimap<double,packet_type> delaySend;
  multimap<double,unsigned int> delaySplit,delayMerge;
  multimap<double,IMBaseOutEntity *> delayRespond;

  unsigned int clientaddress;

public:

  Msocket *msocket;
  static IMBaseClient *gClient;

  IMBaseClient();
  ~IMBaseClient();

  virtual void tick();
  double getTime() { return simtime; }
  void setTime(double t) { simtime = t; }
```

```
   static int numrtt;
   static int getNextRTT() { numrtt++; return numrtt-1; }

   virtual void addOutEntity(IMBaseOutEntity *e) { outlist.insert(e); }
   virtual void remOutEntity(IMBaseOutEntity *e) { outlist.erase(e); }

   virtual void checkin(IMBaseOutEntity *e,unsigned int from,
                        unsigned int to);
   virtual void sendMoved(IMBaseOutEntity *e,McastPacket &mpacket,
                unsigned int send_to,unsigned int to,unsigned int from);
   virtual void sendPing(unsigned int send_to);
   virtual void sendMcast(unsigned int send_to,McastPacket &packet);
   virtual void sendDelayedMcast(unsigned int send_to,
                                 McastPacket &packet,double delay);

   virtual unsigned int OCTREE_search_server(IMBaseOutEntity *e);
   virtual unsigned int OCTREE_find_primary(IMBaseOutEntity *e,
                                            bool ask);
   virtual unsigned int OCTREE_ask_friends(IMBaseOutEntity *e,
                                           unsigned int address);

   virtual bool MDHCP_getAddresses(int num,unsigned int *addrs);
};

#endif
```

2. IMBaseEntity.h

```
#ifndef _IMBASEENTITY
#define _IMBASEENTITY

#include <set.h>

#include "IMPacket.h"

#include "npsVec3f.h"
#include "IMCellRegion.h"

class IMBaseEntity
{
 public:

   unsigned int address; // Server assigned multicast address or guid.

   npsVec3f pos;         // position
   npsVec3f vel;         // velocity

   double lastUpdate;    // Time of last update (secs past Jan 1, 1970)

   CellRegion primary_region; //copy of primary region XXXX why?

 protected:

   npsVec3f oldpos;      // Position at time = lastUpdate

 public:

   enum { INBOUND,OUTBOUND };
   virtual int   mode() = 0;

   virtual int   runtime_type()     { return rtt; }
   virtual char *protocol_name()    { return module_name; }
   virtual char *protocol_url()     { return module_url; }
   virtual float protocol_version() { return module_version; }

   static int rtt;
   static char module_name[64];
   static char module_url[192];
   static float module_version;

   virtual void tick() = 0; // Update Entity

   virtual bool interestedIn(IMBaseEntity *e) { return false; }

   struct compare :
   public binary_function<IMBaseEntity *,IMBaseEntity *,bool>
   {
    bool operator()(IMBaseEntity *r1,IMBaseEntity *r2) const
              {
               return (r1->address < r2->address);
              }
   };
};

#endif
```

3. IMBaseInEntity.h

```
#ifndef _IMBASEINENTITY
#define _IMBASEINENTITY

#include "IMBaseEntity.h"
#include "IMPacket.h"

#include "string"
#include "map.h"

class IMBaseInEntity : public IMBaseEntity
{
 protected:

   int internal;   // ref count; 0 == entity updates come from octree

   double lastlastUpdate;
   npsVec3f oldoldpos;
   npsVec3f oldvel;
   float smooth_time;

   double dt;

   McastPacket lastPacket;

 public:

   IMBaseInEntity() {}

   IMBaseInEntity(McastPacket &mpacket)
   {
    internal = 0;
    address = mpacket.newaddress[1];
    update(mpacket);
   }

   virtual int mode() { return INBOUND; }

   virtual void tick(); // Update Entity
   virtual void update(McastPacket &mpacket);

   virtual bool subscribed() { return (internal > 0); }
   virtual void subscribe() { internal++; }
   virtual void unsubscribe() { internal--; }

   map<string,void *> attributeMap;
};

#endif
```

4. IMBaseOutEntity.h

```
#ifndef _IMBASEOUTENTITY
#define _IMBASEOUTENTITY

#include "npsVec3f.h"
#include "IMBaseEntity.h"
#include "IMBaseProtocol.h"
#include "IMCellRegion.h"

#include "map.h"

class IMNode;

class IMBaseOutEntity : public IMBaseEntity
{
 protected:

   float dead_thresh;          // Update if error > dead_thresh (m^2)
   float dead_timeout;         // Update if time > dead_timeout (s)

   void obtain_address();
   void checkin(unsigned int old);

   unsigned int find_primary();

   npsVec3f lastPos;

 public:


   map<IMBaseEntity *,float,IMBaseEntity::compare> interestingEntities;


 protected:

   double lastTick;                 // Time this guy was ticked last
   double dt;                       // currenttime - lastTick

   void Update_dt();                // Updates dt;
   CellRegion region;   // Current location
   npsVec3f oldvel;  // Velocity at time = lastUpdate

   set<IMNode *> current_nodes; // Set of interesting regions

 public:

   virtual int mode() { return OUTBOUND; }

   virtual void sendMoved(McastPacket &mpacket,unsigned int send_to,
                          unsigned int to,unsigned int from);
   virtual void forceSend(); // Used for pings
   virtual void tick();      // Update Entity

   IMBaseOutEntity(npsVec3f p, float search_r, float dia,
                   float vel,float timeout);
   IMBaseOutEntity(npsVec3f p, float search_r);

   ~IMBaseOutEntity();
};
#endif
```

5. IMBaseNode.h

```
#ifndef _IMBASENODE
#define _IMBASENODE

#include "set.h"
#include "map.h"

#include "IMCellRegion.h"

//Forward declare classes
class IMBaseOutEntity;

class IMNode
{
 public:
   static map<unsigned int,IMNode *> nodemap;

   CellRegion region; //Bounds
   IMNode *parent;
   IMNode *child[8];
   bool leaf;         //True if node has no children
   int ref;

   set<unsigned int> elist;

   IMNode(CellRegion r)
   {
    parent=NULL;
    leaf=true;
    for(int i=0;i<8;i++) child[i]=NULL;
    memcpy(&region,&r,sizeof(CellRegion));
    ref=0;
   }

   IMNode()
   {
    parent=NULL;
    leaf=true;
    for(int i=0;i<8;i++) child[i]=NULL;
    ref=0;
   }

   virtual bool addtotree();
   virtual bool remfromtree();
   virtual IMNode *XYZtonode(IMBaseOutEntity *e);
   virtual int searchXYZR(double x,double y,double z,double r,
                          CellRegion *cells);

};

#endif
```

6. IMBaseProtocol.h

```
#ifndef __IMBASEPROTOCOL__
#define __IMBASEPROTOCOL__

#include "bbMappedClass.h"

#include "IMPacket.h"

// forward declare
class IMBaseInEntity;

class IMBaseProtocol : public bbMappedClass<IMBaseProtocol>
{
 public:

   IMBaseProtocol(): bbMappedClass<IMBaseProtocol>("IMBaseClientModule")
                    {setup();}
   IMBaseProtocol(const char *name): bbMappedClass<IMBaseProtocol>(name)
                    {setup();}

   virtual void setup(void);

   virtual IMBaseInEntity *new_entity(McastPacket &mpacket);
   virtual void del_entity(IMBaseInEntity *);
};

class IMBaseProtocolIE
{
 public:
   IMBaseProtocol *protocol; //Which protocol is this for

   struct compare :
   public binary_function<IMBaseProtocolIE *,IMBaseProtocolIE *,bool>
   {
    bool operator()(IMBaseProtocolIE *r1,IMBaseProtocolIE *r2) const
                {
                 return (r1->protocol < r2->protocol);
                }
   };
};
#endif
```

7. IMCellRegion.h

```
#ifndef __IM_CELLREGION
#define __IM_CELLREGION

#include "npsVec3f.h"
#include "IMExtent.h"

struct CellRegion
{
  unsigned int address;
  Extent ext;
  int below;

  inline bool contains(npsVec3f p)
  {
   if (p[0] <= ext.x + ext.r && p[0] > ext.x - ext.r &&
       p[1] <= ext.y + ext.r && p[1] > ext.y - ext.r &&
       p[2] <= ext.z + ext.r && p[2] > ext.z - ext.r) return true;
   return false;
  }

  inline bool contains(npsVec3f p,float r)
  {
   r+=ext.r;
   if (p[0] <= ext.x + r && p[0] > ext.x - r &&
       p[1] <= ext.y + r && p[1] > ext.y - r &&
       p[2] <= ext.z + r && p[2] > ext.z - r) return true;
   return false;
  }
};

#endif
```

8. IMExtent.h

```
#ifndef __IM_EXTENT
#define __IM_EXTENT

// Messages from server
struct Extent
{
 double x;
 double y;
 double z;
 double r;
};

#endif
```

9. IMPacket.h

```
#ifndef __IM_PACKET
#define __IM_PACKET

#include "IMCellRegion.h"

// Messages to server
struct ServerPacket
```

```
{
 enum PacketType {SEARCH,CHANGE,MALLOC,LOCK,SPLIT,MERGE};
 enum PacketType type;
 union
 {
  struct
  {
   int num;
  } malloc;

  struct
  {
   unsigned int address;
   unsigned int newaddr[8];
  } split;

  struct
  {
   unsigned int address;
   double x;
   double y;
   double z;
   double r;
  } search;

  struct
  {
   unsigned int from;
   unsigned int to;
  } change;

 };
};

// Messages to mcast address
#define MIN_PACKET_SIZE 312
struct McastPacket
{
 enum PacketType {MOVED,PINGING,MERGE,SPLIT,QUERY,RESPOND};
 enum PacketType type;
 unsigned int address;
 union
     {
      unsigned int newaddress[8]; //XXXX is this used?!
      struct
           {
            unsigned int newaddress[2];
            double lastUpdate;
            double x;
            double y;
            double z;
            float vx;
            float vy;
            float vz;
            char module_name[64];
            char module_url[192];
            float module_version;
            char protocol_reserved[712];
           } moved; // 312 + 712 = 1024

      struct
```

```
            {
             unsigned int tcpaddress;
             unsigned int port;
             double r;
             double x;
             double y;
             double z;
            } query; // 40 bytes
      };
};

#endif
```

10. IMSocket.h

```
#ifndef _MSOCKET
#define _MSOCKET

#include <set.h>

#include "ace/SOCK_Dgram.h"

#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */

#include "ace/INET_Addr.h"

//A Socket that reads from multiple mcast addresses

class ACE_Export Msocket : public ACE_SOCK_Dgram
{
 public:
   Msocket(unsigned short port);
   ~Msocket(void);

   bool subscribe(unsigned int address);
   void unsubscribe(unsigned int address);

 private:
   set<unsigned int> sub_list;
   in_addr ifaddr;
};

#endif
```

11. baseClient.c++

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#include <sys/time.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "bbPrinter.h"
```

```
#include "bbThread.h"

#include "IMBaseClient.h"
#include "IMBaseOutEntity.h"
#include "IMBaseInEntity.h"
#include "IMBaseProtocol.h"

#include "IMSocket.h"
#include "ace/Handle_Set.h"

#include "IMPacket.h"
#include "IMBaseNode.h"

#include "map.h"

#define JOIN_DELAY 10

int IMBaseClient::numrtt = 0;
IMBaseClient *IMBaseClient::gClient;

IMBaseClient::IMBaseClient()
{
 internal_clock = true;
 auto_tick = true;

 struct timeval tt;
 gettimeofday(&tt);
 setTime((double)tt.tv_sec + tt.tv_usec/1000000.0);
 outlist.clear();
 inlist.clear();
 setup();

 //Turn on the base protocol. This is our default entity.
 base_proto = new IMBaseProtocol();
 IMBaseEntity::rtt = IMBaseClient::getNextRTT();

 //Start up the network read thread
 readThread = new bbThread(readFunc,NULL);
}

void IMBaseClient::setup()
{
 // Setup the regions
  for(int x=0;x<WORLDSIZE*2;x++)
    {
     for(int y=0;y<WORLDSIZE*2;y++)
        {
         for(int z=0;z<WORLDSIZE*2;z++)
            {
             static unsigned int addr = inet_addr("239.0.0.1");
             CellRegion r;
             r.address = addr;
             r.ext.x = REGION_SIZE*x + REGION_SIZE/2.0 -
                       WORLDSIZE*REGION_SIZE;
             r.ext.y = REGION_SIZE*y + REGION_SIZE/2.0 -
                       WORLDSIZE*REGION_SIZE;
             r.ext.z = REGION_SIZE*z + REGION_SIZE/2.0 -
                       WORLDSIZE*REGION_SIZE;
             r.ext.r = REGION_SIZE/2.0;
             tree[x][y][z] = new IMNode(r);
             IMNode::nodemap.insert(pair<unsigned int,IMNode *>
                                     (addr,tree[x][y][z]));
             addr++;
            }
        }
    }

 // Set up the multicast socket
 int on=1,ttl = 15;
 unsigned char off = 0;
 msocket = new Msocket(9976);
 s2 = socket(AF_INET, SOCK_DGRAM, 0);
 setsockopt(s2, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
 if (setsockopt(s2, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on)) < 0)
    perror("setsockopt SO_REUSEPORT");
 if (setsockopt(s2, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
    perror("setsockopt SO_REUSEADDR");
 if (setsockopt(s2, IPPROTO_IP, IP_MULTICAST_LOOP, &off, sizeof(off))<0)
    perror("setsockopt IP_MULTICAST_LOOP");

 // Set up our tcp socket
 struct in_addr bind_address;
 bind_address.s_addr = htonl(INADDR_ANY);
}


IMBaseClient::~IMBaseClient()
{
 printf("IMBaseClient shutting down\n");
 delete readThread;
 printf("Killed readThread.\n");
 delete msocket;
 printf("deleted msocket.\n");
 printf("IMBaseClient shutdown\n");
}

void IMBaseClient::tick()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 struct timeval tt;
 gettimeofday(&tt);
 double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;

 // Go through delay send queue
 multimap<double,packet_type>::iterator p = delaySend.begin();
 while(p != delaySend.end() && (*p).first < tod)
      {
       sendMcast((*p).second.first,(*p).second.second);
       delaySend.erase(p);
       p = delaySend.begin();
      }

 //Set simtime based on tod
 if (internal_clock) setTime(tod);

 set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i;
 set<IMBaseInEntity *,IMBaseEntity::compare>::iterator j;

 // This manages the interestingEntities list for inbounds.
 for(j = inlist.begin();j!=inlist.end();j++)
    for(i = outlist.begin();i!=outlist.end();i++)
```

```
        {
         float dist2 = ((*i)->pos - (*j)->pos).lengthSqr();
         if (dist2 < (*i)->roi2 && (*i)->interestedIn(*j))
            (*i)->interestingEntities.insert(pair<IMBaseEntity *,float>
                                            (*j,dist2));
         else
            (*i)->interestingEntities.erase(*j);
        }

 // This manages the interestingEntities list for outbounds.
 for(i = outlist.begin();i!=outlist.end();i++)
    {
     set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i2 = i;
     for(i2++;i2!=outlist.end();i2++)
        {
         float dist2 = ((*i)->pos - (*i2)->pos).lengthSqr();
         if (dist2 < (*i)->roi2 && (*i)->interestedIn(*i2))
             (*i)->interestingEntities.insert(pair<IMBaseEntity *,float>
                                             (*i2,dist2));
         else
             (*i)->interestingEntities.erase(*i2);
         if (dist2 < (*i2)->roi2 && (*i2)->interestedIn(*i))
             (*i2)->interestingEntities.insert(pair<IMBaseEntity *,float>
                                              (*i,dist2));
         else
             (*i2)->interestingEntities.erase(*i);
        }
    }

 if (auto_tick)
    {
     for(i = outlist.begin();i!=outlist.end();i++)
        (*i)->tick();

     for(j = inlist.begin();j!=inlist.end();j++)
        (*j)->tick();
    }
}

void IMBaseClient::checkin(IMBaseOutEntity *e,unsigned int from,unsigned
int to)
{
if (from != to)
  {
   static McastPacket p;
   e->sendMoved(p,from,to,from);
   e->sendMoved(p,to,to,from);
  }
}

void IMBaseClient::sendDelayedMcast(unsigned int send_to,McastPacket
&packet,double delay)
{
 struct timeval tt;
 gettimeofday(&tt);
 double tod = (double)tt.tv_sec + tt.tv_usec/1000000.0;
 packet_type packetPair(send_to,packet);
 multimap<double,packet_type>::value_type
          valuePair(delay+tod,packetPair);
 delaySend.insert(valuePair);
}
```

```
void IMBaseClient::sendMcast(unsigned int send_to,McastPacket &packet)
{
 struct sockaddr_in sa_mcast;
 sa_mcast.sin_family=AF_INET;
 struct in_addr grpaddr;
 grpaddr.s_addr = send_to;
 sa_mcast.sin_addr=grpaddr;
 sa_mcast.sin_port=htons(9976);

 sendto(s2,&packet,sizeof(McastPacket),0,&sa_mcast,sizeof(sa_mcast));
}

void IMBaseClient::sendPing(unsigned int send_to)
{
 McastPacket packet;
 packet.type=McastPacket::PINGING;
 packet.address=send_to;

 sendDelayedMcast(send_to,packet,JOIN_DELAY);
}

void IMBaseClient::sendMoved(IMBaseOutEntity *e,McastPacket
&mpacket,unsigned int send_to,unsigned int to,unsigned int from)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);
 if (to != from)
    {
     IMNode *n = tree[0][0][0]->findnode(from);
     if (n)
        n->elist.erase(e->address);
     n = tree[0][0][0]->findnode(to);
     if (n)
        n->elist.insert(e->address);
    }

 sendMcast(send_to,mpacket);
}

unsigned int IMBaseClient::OCTREE_find_primary(IMBaseOutEntity *e,bool
ask)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

//Check to see if we are currently in the correct region.
 if (e->primary_region.address != 0)
    if (e->primary_region.contains(e->pos))
       return 0;

 int x = (int)(e->pos[0]/REGION_SIZE) + WORLDSIZE;
 int y = (int)(e->pos[1]/REGION_SIZE) + WORLDSIZE;
 int z = (int)(e->pos[2]/REGION_SIZE) + WORLDSIZE;
 IMNode *tmp = tree[x][y][z];

 if (tmp) // If we found an answer
    {
     unsigned int rval = e->primary_region.address;
     // Work around bug in 'e->primary_region.contains', I think...
     if (rval == tmp->region.address) return 0;
     memcpy(&e->primary_region,&tmp->region,sizeof(CellRegion));
     return rval;
    }
 printf("You should never see this message!\n");
 return 0;
}

void IMBaseClient::filter(McastPacket &mpacket,IMBaseInEntity *ng)
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 //Does Protocol Exist? O(lg P)
 IMBaseProtocol *protocol =
                 IMBaseProtocol::findObject(mpacket.moved.module_name);
 if (protocol == NULL)
    {
     bbModule::load(mpacket.moved.module_name,
               mpacket.moved.module_version,0,mpacket.moved.module_url);
     protocol = IMBaseProtocol::findObject(mpacket.moved.module_name);

     if (protocol == NULL)
        {
         protocol = base_proto;
        }
    }

 //Does Entity Exist?
 set<IMBaseInEntity *,IMBaseEntity::compare>::iterator i;

 // Yes - an old guy. O(lg N)
 if ((i = inlist.find(ng)) != inlist.end())
    {
     // Update his packet unless we getting better updates elsewhere?
     if ((*i)->subscribed()) return;
     else
        if ((*i)->lastUpdate < mpacket.moved.lastUpdate)
           (*i)->update(mpacket);
    }
 else // No - a new guy O(lg N)
    {
     i = inlist.insert(protocol->new_entity(mpacket)).first;
    }

 // If he is checking out, delete him!
 if (mpacket.address == 0)
    {
     printf("Killing entity...\n");
     IMBaseInEntity *tmp = *i;
     inlist.erase(i);
     protocol->del_entity(tmp);
    }
 else
 //Keep track of number and who is in each region.
 //Atempt to split or merge if needed.
 if ((*i)->primary_region.address != mpacket.address)
    {
     IMNode *n = tree[0][0][0]->findnode((*i)->primary_region.address);
     if (n)
        n->elist.erase((*i)->address);
     n = tree[0][0][0]->findnode(mpacket.address);
     if (n)
        n->elist.insert((*i)->address);
    }

}

void IMBaseClient::readFunc(bbThread *,bbData *that)
{
//printf("IMBaseClient::readFunc\n");

 //XXXX avoid race condition
 if (IMBaseClient::gClient != NULL)
    IMBaseClient::gClient->mcastcheck();
 else
    sleep(0);
}

void IMBaseClient::mcastcheck()
{
 ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

 int rval;
 static McastPacket mpacket;
 ACE_Handle_Set handle_set;

 handle_set.reset();
 handle_set.set_bit(msocket->get_handle());

 // Time out so thread will unblock.
 ACE_Time_Value t2(1,0);

 if ((rval = ACE_OS::select((msocket->get_handle())+1,handle_set,
                             NULL,NULL,&t2)) != 0)
    {
     if (rval == -1) perror("select");
     else
        {
         ACE_OS::recvfrom(msocket->get_handle(),(char *)&mpacket,
                          sizeof(McastPacket), 0,0,0);
         switch (mpacket.type)
                {
                 case McastPacket::MOVED:
                          {
                           static IMBaseInEntity *ng =
                                       new IMBaseInEntity(mpacket);
                           ng->address = mpacket.newaddress[1];
                           filter(mpacket,ng);
                          }
                          break;

                 case McastPacket::PINGING:
                          {
               printf("Got Ping!\n");
               set<IMBaseOutEntity *,IMBaseEntity::compare>::iterator i;
                 for(i = outlist.begin();i!=outlist.end();i++)
                    if ((*i)->primary_region.address == mpacket.address)
                       (*i)->forceSend();

 // Remove ping request if we already go one.
 multimap<double,packet_type>::iterator p = delaySend.begin();
 for(;p != delaySend.end(); p++)
    {
     if ((*p).second.first == mpacket.address &&
         (*p).second.second.type == McastPacket::PINGING)
        {
         delaySend.erase(p);
         printf("Ping request removed..\n");
         break;
        }
    }

                          }
                          break;

                 default: printf("Unknown message from socket!\n");
                }
        }
    }
}
```

12. baseInEntity.c++

```
#include "IMBaseEntity.h"

int IMBaseEntity::rtt;
char IMBaseEntity::module_name[64] = "IMBaseClientModule";
char IMBaseEntity::module_url[192] = "";
float IMBaseEntity::module_version = 0.894;

#include "IMBaseClient.h"
#include "IMBaseInEntity.h"

void IMBaseInEntity::update(McastPacket &mpacket)
{
 memcpy(&lastPacket,&mpacket,sizeof(McastPacket));
 npsVec3f newpos(mpacket.moved.x,mpacket.moved.y,mpacket.moved.z);
 npsVec3f newvel(mpacket.moved.vx,mpacket.moved.vy,mpacket.moved.vz);

 oldoldpos = oldpos;
 oldpos = newpos;

 oldvel = vel;
 vel = newvel;

 lastlastUpdate = lastUpdate;
 lastUpdate = mpacket.moved.lastUpdate;
}

void IMBaseInEntity::tick()
{
 dt = IMBaseClient::gClient->getTime()-lastUpdate;
 pos = oldpos + vel * dt;

 if (dt < smooth_time)
    {
     double dt2 = IMBaseClient::gClient->getTime()-lastlastUpdate;
     float frac = dt/smooth_time;
     pos = pos * frac + (oldoldpos + oldvel * dt2) * (1.0f-frac);
    }
}
```

13. baseNode.c++

```
#include "IMSocket.h"
#include "IMBaseNode.h"
#include "IMBaseClient.h"
#include "IMBaseInEntity.h"
#include "IMBaseOutEntity.h"

map<unsigned int,IMNode *> IMNode::nodemap;

IMNode *IMNode::findnode(unsigned int address)
{
 map<unsigned int,IMNode *>:: iterator i = nodemap.find(address);

 if (i != nodemap.end()) return (*i).second;

 return NULL;
}

bool IMNode::remfromtree()
{
 ref--;

 if (ref > 0) return false;

 IMBaseClient::gClient->msocket->unsubscribe(region.address);

 set<unsigned int>::iterator i;
 for(i= elist.begin();i != elist.end(); i++)
    {
     IMBaseInEntity ng;
     ng.address = *i;
     set<IMBaseInEntity *>::iterator
           i = IMBaseClient::gClient->inlist.find(&ng);
     if (i != IMBaseClient::gClient->inlist.end())
        {
         IMBaseClient::gClient->inlist.erase(i);
         delete *i;
        }
    }

 return true;
}

bool IMNode::addtotree()
{
 ref++;

 if (IMBaseClient::gClient->msocket->subscribe(region.address))
    {
     IMBaseClient::gClient->sendPing(region.address);
printf("Sent ping to %d\n",region.address);
     return true;
    }

 return false;
}

IMNode *IMNode::XYZtonode(IMBaseOutEntity *e)
{
 if (region.contains(e->pos) && e->smallest_region < region.ext.r)
    {
     for(int i=0;i<8;i++)
        {
         if (child[i])
            {
             IMNode *rval = child[i]->XYZtonode(e);
             if (rval) return rval;
            }
        }
     return this;
    }
 return NULL;
}

int IMNode::searchXYZR(double x,double y,double z,double r,CellRegion
*cells)
{
 //Always return parent.
 CellRegion *origregs = cells;
 cells[0].ext.x = region.ext.x;
 cells[0].ext.y = region.ext.y;
 cells[0].ext.z = region.ext.z;
 cells[0].ext.r = region.ext.r;
 cells[0].address = region.address;
 cells = &cells[1];
 int count = 1;

 // Out of the eight child, which do we intersect?
 for(int i=0;i<8;i++)
    {
     if (child[i])
        {
         if (((x-region.ext.x)*(x-region.ext.x) +
              (y-region.ext.y)*(y-region.ext.y) +
              (z-region.ext.z)*(z-region.ext.z)) <
              (2*r*r + 2*region.ext.r*region.ext.r))
            {
             int c = child[i]->searchXYZR(x,y,z,r,cells);
             cells = &cells[c];
             count += c;
            }
        }
    }

 //Set number of regions below this one.
 origregs->below = count-1;

 //Return the number of regions.
 return count;
}
```

14. baseOutEntity.c++

```
#include "IMBaseClient.h"
#include "IMBaseOutEntity.h"
#include "IMBaseNode.h"

// For set_difference
```

```
#include "algo.h"
#include "set.h"
#include "vector.h"

IMBaseOutEntity::IMBaseOutEntity(npsVec3f p, float search_r, float dia,
float vel, float timeout)
{
 roi = search_r;
 roi2 = roi*roi;
 primary_region.address = 0;
 address = 0;

 AvgBoundDia(dia);
 AvgMaxVel(vel);

 dead_timeout = timeout;
 oldvel.makeNull();
 oldpos = pos = p;
 oldvel[0] = 0.0;  oldvel[1] = 0.0;  oldvel[2] = 0.0;
 obtain_address();
 IMBaseClient::gClient->addOutEntity(this);
 lastTick = IMBaseClient::gClient->getTime();
}

IMBaseOutEntity::IMBaseOutEntity(npsVec3f p,float search_r)
{
 roi = search_r;
 roi2 = roi*roi;
 primary_region.address = 0;
 address = 0;

 AvgBoundDia(1.0);
 AvgMaxVel(1.0);

 dead_timeout = 5.0;
 oldvel.makeNull();
 oldpos = pos = p;
 oldvel[0] = 0.0;  oldvel[1] = 0.0;  oldvel[2] = 0.0;
 obtain_address();
 lastTick = IMBaseClient::gClient->getTime();
 IMBaseClient::gClient->addOutEntity(this);
}

IMBaseOutEntity::~IMBaseOutEntity()
{
 IMBaseClient::gClient->remOutEntity(this);
 unsigned int old = primary_region.address;
 primary_region.address = 0;
 checkin(old);
}

void IMBaseOutEntity::obtain_address()
{
 static unsigned int clientaddress = 0;
 if (!clientaddress)
     {
      char myname[255];
      gethostname(myname,255);
      struct hostent *en = gethostbyname(myname);
      clientaddress = *((int *)(en->h_addr_list[0]));
      clientaddress = (clientaddress >> 24) + (clientaddress<<24) +
                          (clientaddress&0x00ffff00);
     }

  address = clientaddress++;
 }

 void IMBaseOutEntity::checkin(unsigned int old)
 {
  IMBaseClient::gClient->checkin(this,old,primary_region.address);
 }

 void IMBaseOutEntity::sendMoved(McastPacket &mpacket,unsigned int
 send_to,unsigned int to,unsigned int from)
 {
  mpacket.type=McastPacket::MOVED;
  mpacket.address=to;
  mpacket.newaddress[0]=from;
  mpacket.newaddress[1]=address;
  mpacket.moved.x = pos[0];
  mpacket.moved.y = pos[1];
  mpacket.moved.z = pos[2];
  mpacket.moved.vx = vel[0];
  mpacket.moved.vy = vel[1];
  mpacket.moved.vz = vel[2];
  mpacket.moved.lastUpdate = lastUpdate;
  mpacket.moved.module_version = protocol_version();
  strcpy(mpacket.moved.module_name,protocol_name());
  strcpy(mpacket.moved.module_url,protocol_url());

  IMBaseClient::gClient->sendMoved(this,mpacket,send_to,to,from);
 }

 unsigned int IMBaseOutEntity::find_primary()
 {
  ACE_Guard<bbMutex> guard(IMBaseClient::gClient->tree_mutex);

  set<IMNode *> new_nodes;

  // We end up doing this twice cause entitiy doesn't have an IMNode *...
  int x = (int)(pos[0]/REGION_SIZE) + WORLDSIZE;
  int y = (int)(pos[1]/REGION_SIZE) + WORLDSIZE;
  int z = (int)(pos[2]/REGION_SIZE) + WORLDSIZE;
  IMNode *tmp = IMBaseClient::gClient->tree[x][y][z];

  if (tmp)
     {
      new_nodes.insert(tmp);

      if (x == 0) x = 1;
      if (y == 0) y = 1;
      if (z == 0) z = 1;

      if (x == 2*WORLDSIZE) x = 2*WORLDSIZE - 1;
      if (y == 2*WORLDSIZE) y = 2*WORLDSIZE - 1;
      if (z == 2*WORLDSIZE) z = 2*WORLDSIZE - 1;

      for(int xx = x-1; xx < x+2; xx++)
         for(int yy = y-1; yy < y+2; yy++)
            for(int zz = z-1; zz < z+2; zz++)
               {
                IMNode *n = IMBaseClient::gClient->tree[xx][yy][zz];
               if (n->region.contains(pos,roi))
                  new_nodes.insert(n);
              }

     vector<IMNode *> add_nodes(new_nodes.size(),NULL);
     vector<IMNode *>::iterator a = add_nodes.begin();
     a = set_difference(new_nodes.begin(),new_nodes.end(),
                        current_nodes.begin(),current_nodes.end(),a);
     for(a = add_nodes.begin();a != add_nodes.end(); a++)
        if (*a)
           {
            c++;
            (*a)->addtotree();
            current_nodes.insert(*a);
           }

     vector<IMNode *> old_nodes(current_nodes.size(),NULL);
     vector<IMNode *>::iterator o = old_nodes.begin();
     o = set_difference(current_nodes.begin(),current_nodes.end(),
                        new_nodes.begin(),new_nodes.end(),o);
     for(o = old_nodes.begin() ;o != old_nodes.end(); o++)
        if (*o)
           {
            c++;
            (*o)->remfromtree();
            current_nodes.erase(*o);
           }
    }

 return IMBaseClient::gClient->OCTREE_find_primary(this,true);
}

void IMBaseOutEntity::Update_dt()
{
 dt = IMBaseClient::gClient->getTime() - lastTick;
}

void IMBaseOutEntity::forceSend()
{
 double time = IMBaseClient::gClient->getTime();
 lastUpdate = time;
 oldvel = vel;
 oldpos = pos;
 static McastPacket p;
 sendMoved(p,primary_region.address,primary_region.address,0);
}

void IMBaseOutEntity::tick()
{
 double time = IMBaseClient::gClient->getTime();
 double big_dt = time-lastUpdate;

 int oldcell = find_primary();

 npsVec3f ghostpos = oldpos + oldvel * big_dt;
 vel = (pos - lastPos)/dt;

 if ((ghostpos - pos).lengthSqr() > dead_thresh ||
      big_dt > dead_timeout || oldcell)
    {
     lastUpdate = time;
     oldvel = vel;
     oldpos = pos;
     static McastPacket p;
     if (oldcell) checkin(oldcell);
     else sendMoved(p,primary_region.address,primary_region.address,0);
    }

 lastPos = pos;
 lastTick = time;
}

void IMBaseOutEntity::calc_smallest_region()
{
 dead_thresh = 1.0;
 printf("dead_thresh now %f meters.\n",dead_thresh);
}
```

15. baseProtocol.c++

```
#include "IMBaseClient.h"
#include "IMBaseProtocol.h"
#include "IMBaseInEntity.h"

IMBaseInEntity *IMBaseProtocol::new_entity(McastPacket &mpacket)
{
 return new IMBaseInEntity(mpacket);
}

void IMBaseProtocol::del_entity(IMBaseInEntity *e)
{
 delete e;
}

void IMBaseProtocol::setup(void)
{
}
```

16. msocket.c++

```
#include "IMSocket.h"

Msocket::Msocket(unsigned short port)
{
 if (ACE_SOCK_Dgram::open(ACE_INET_Addr(port),PF_INET,0,1) < 0)
    perror("open:");

 char buf[BUFSIZ];
 struct ifconf ifc;
 ifc.ifc_len = sizeof(buf);
 ifc.ifc_buf = buf;
 if (ACE_OS::ioctl(get_handle(), SIOCGIFCONF, &ifc) < 0)
    perror("ioctl SIOCGIFCONF");
 struct ifreq *ifr;
 ifr = ifc.ifc_req;
 ifaddr = ((struct sockaddr_in *)&ifr->ifr_addr)->sin_addr;

 int on = 1;
 if (set_option(SOL_SOCKET,SO_REUSEADDR,&on,sizeof on) == -1)
```

```
    perror("setsockopt SO_REUSEADDR");
 if (set_option(SOL_SOCKET,SO_REUSEPORT,&on,sizeof on) == -1)
    perror("setsockopt SO_REUSEPORT");

 int off = 0;
 if (set_option(IPPROTO_IP, IP_MULTICAST_LOOP, &off, sizeof off) == -1)
    perror("msocket setsockopt IP_MULTICAST_LOOP");
}

Msocket::~Msocket()
{
 ACE_SOCK_Dgram::close();
}

bool Msocket::subscribe(unsigned int address)
{
 //Don't bother if already subscribed
 if (!sub_list.insert(address).second) return false;

 ip_mreq mreq;
 mreq.imr_multiaddr.s_addr = address;
 mreq.imr_interface = ifaddr;
 if (ACE_SOCK::set_option(IPPROTO_IP,IP_ADD_MEMBERSHIP,&mreq,sizeof
mreq)== -1)
    perror("setsockopt add membership");

 return true;
}

void Msocket::unsubscribe(unsigned int address)
{
 //Don't bother if not already subscribed
 if (!sub_list.erase(address)) return;

 in_addr grpaddr;
 ip_mreq mreq;
 grpaddr.s_addr = address;
 mreq.imr_multiaddr = grpaddr;
 mreq.imr_interface = ifaddr;
 if (ACE_SOCK::set_option(IPPROTO_IP,IP_DROP_MEMBERSHIP,&mreq,
     sizeof mreq)== -1)
    perror("setsockopt add membership");
}
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, Virginia 22060-6218

2. Dudly Knox Library 2
Naval Postgraduate School
411 Dyer Rd.
Monterey, California 93943-5101

3. Dr. Don Brutzman 1
Code UW/BR
Naval Postgraduate School
Monterey, California 93943-5001

4. Dr. Rudy Darken 2
Code CS
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5001

5. Dr. Ted Lewis 1
Code CS
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5001

6. Dr. Sandeep Singhal 2
IBM Corporation
P.O. Box 12195
RTP, NC 27709

7. Dr. Michael J. Zyda 2
Code CS/ZK
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5001

8. Dr. Michael Macedonia 1
Chief Scientist and Technical Director
US Army STRICOM
12350 Research Parkway
Orlando, Florida 32826-3276

9. Jaron Lanier 1
Advanced Network & Services, Inc.
200 Business Park Drive
Armonk, New York 10504

10. Al Weis 1
Advanced Network & Services, Inc.
200 Business Park Drive
Armonk, New York 10504

11. Michael Myjak 1
1615 South Carpenter Road
Titusville, Florida 32796

12. Dr. Mark Pullen 1
Computer Science/C3I Center MS4A5
George Mason University
Fairfax, Virginia 22030

13. Nagesh Kakarlamudi 1
9577 Sunnyslope Dr.
Manassas, Virginia 20112-2766

14. Mr & Mrs Abrams 1
1339 Green Knolls Drive
Buffalo Grove, Illinois 60089

15. Dr. Howard Abrams 4
Code CS
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5001